Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations

LENNARD GÄHER, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
SIMON SPIES, MPI-SWS, Germany
RALF JUNG, MPI-SWS, Germany
HOANG-HAI DANG, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
JEEHOON KANG, KAIST, Korea
DEREK DREYER, MPI-SWS, Germany

Today's compilers employ a variety of non-trivial optimizations to achieve good performance. One key trick compilers use to justify transformations of *concurrent* programs is to assume that the source program has no *data races*: if it does, they cause the program to have *undefined behavior* (UB) and give the compiler free rein. However, verifying correctness of optimizations that exploit this assumption is a non-trivial problem. In particular, prior work either has not proven that such optimizations preserve program termination (particularly non-obvious when considering optimizations that move instructions out of loop bodies), or has treated all synchronization operations as external functions (losing the ability to reorder instructions around them).

In this work we present *Simuliris*, the first simulation technique to establish termination preservation (under a fair scheduler) for a range of concurrent program transformations that exploit UB in the source language. Simuliris is based on the idea of using *ownership* to reason modularly about the assumptions the compiler makes about programs with well-defined behavior. This brings the benefits of *concurrent separation logics* to the space of verifying program transformations: we can combine powerful reasoning techniques such as framing and coinduction to perform *thread-local* proofs of non-trivial concurrent program optimizations. Simuliris is built on a (non-step-indexed) variant of the Coq-based Iris framework, and is thus not tied to a particular language. In addition to demonstrating the effectiveness of Simuliris on standard compiler optimizations involving data race UB, we also instantiate it with Jung et al.'s Stacked Borrows semantics for Rust and generalize their proofs of interesting type-based aliasing optimizations to account for concurrency.

28

CCS Concepts: • **Theory of computation** → **Separation logic**.

Additional Key Words and Phrases: separation logic, program optimizations, data races, Iris

ACM Reference Format:
Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (January 2022), 31 pages. https://doi.org/10.1145/3498689

Authors' addresses: Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Ralf Jung, MPI-SWS, Saarland Informatics Campus, Germany, jung@mpi-sws.org; Hoang-Hai Dang, MPI-SWS, Saarland Informatics Campus, Germany, haidang@mpi-sws.org; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, robbert@cs.ru.nl; Jeehoon Kang, KAIST, Korea, jeehoon.kang@kaist.ac.kr; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2022/1-ART28
https://doi.org/10.1145/3498689

1 INTRODUCTION

Modern compilers use many non-trivial optimizations to make programs run faster. As an example, consider the following simple function, which sums up the value of `*y` until a counter reaches `*x`:

```
int foo(int *x, int *y) {

    int i = 0; int sum = *y;
    while (i != *x) {
        i += 1; sum += *y;
    }
    return sum;
}
```

```
int foo_optimized(int *x, int *y) {
    int n = *x; int m = *y;
    int i = 0; int sum = m;
    while (i != n) {
        i += 1; sum += m;
    }
    return sum;
}
```

When passed the `-O2` flag, Clang optimizes the body of this function to roughly the equivalent of the code on the right.[1] The compiler has moved all the pointer loads out of the loop, so it can access memory once and then keep the values in registers.

Why is this transformation correct? In a sequential language, this is rather trivial, but if concurrency needs to be considered then this optimization might seem incorrect: if another thread changes first `x`, then `y` while `foo` is running, the optimized program may only use the updated value of `y`, while the unoptimized program would also subsequently see the updated value of `x`. Thus the optimized program can produce results not possible for the original program. The reason this optimization is correct in languages like C, C++, and Rust is that unsynchronized concurrent accesses (usually called *data races*) are *undefined behavior* (UB), which means the compiler may assume that no such accesses happen. Under this assumption, no other thread may write to `x` or `y`, thus validating the optimization.

Correctness of program transformations. The goal of our work is to formally establish correctness of optimizations such as our motivating example. There is a lot of prior work on verifying program transformations, which we briefly discuss to be able to explain the new aspects our work brings to this space. An overview can be found in Figure 1.

When it comes to verifying program optimizations, the most obvious points of comparison are verified optimizing compilers such as CompCert [Leroy 2006, 2009] and CakeML [Kumar et al. 2014]. Both of these flagship projects have verified correctness of a number of non-trivial optimization passes. However, CakeML handles only sequential programs, so data races are not considered.

CompCert has seen many extensions with support for concurrency. CompCertTSO [Ševčík et al. 2013] uses the TSO (total store order) memory model, which does not treat data races as UB, thus ruling out optimizations such as the one above. CASCompCert [Jiang et al. 2019] enriches the sequential semantics with a notion of "footprints" such that correctness of optimizations on the sequential language implies correctness in a concurrent context. A similar approach is also used by the "Concurrent CompCert" line of work [Beringer et al. 2014; Cuellar 2020]. This approach can in principle handle our motivating example, though CompCert does not actually perform such transformations. However, it requires hiding all synchronizing operations behind external function calls ("FFI") in the sequential semantics; this rules out most optimizations that reorder instructions around such synchronizing operations, such as the one we consider in §3.2. The same applies to CCAL [Gu et al. 2018], which comes with a thread-safe variant of CompCertX [Gu et al. 2015]: here, *all* accesses to shared memory (not just synchronizing operations) are treated as external function calls and thus are not subject to optimizations.

[1] It then actually goes further and replaces the entire loop by a multiplication.

	DR Opt.	Concurrent	(Fair) TP	Mech.	SL
Ševčík [2009], Morisset et al. [2013]	●	●	○	○	○
Vafeiadis et al. [2015]	●	●	○	●	○
CompCertTSO [Ševčík et al. 2013]	○	●	◐ (no fairness)	●	○
CAS/Concurrent CompCert	◐ (NA)	◐ (FFI)	◐ (no fairness)	●	○
CCAL (CompCertX) [Gu et al. 2018]	○	◐ (FFI)	●	●	○
Liang and Feng [2016]	○	●	●	○	●
ReLoC [Frumin et al. 2018]	○	●	○	●	●
Tassarotti et al. [2017]	○	●	◐ (bounded)	●	●
Transfinite Iris [Spies et al. 2021]	○	○	◐ (sequential)	●	●
Stacked Borrows [Jung et al. 2020]	○	○	◐ (sequential)	●	◐ (model)
Simuliris	●	●	●	●	●

Fig. 1. Comparison of related work (DR Opt = optimizations exploiting UB of data races; NA = only for reordering non-atomics around non-atomics; FFI = only via external functions; TP = termination preservation; Mech = Mechanized; SL = Separation Logic).

Fair termination preservation. The first work to verify transformations which leverage undefined behavior of data races is found in Ševčík's PhD thesis [Ševčík 2009, 2011]. However, they do not consider the same notion of correctness as the CompCert variants we have discussed so far: Ševčík only considers finite traces, and as such does not establish *termination preservation*. The same limitation applies to later work on optimizations for C(-like) memory models [Morisset et al. 2013; Vafeiadis et al. 2015]. Termination preservation ensures that the compiler is not allowed to turn a terminating program into a diverging one. Such a transformation would not be considered "correct" in practice, making termination preservation a key part of a compiler's correctness condition.

However, when considering concurrent programs, even termination-preserving compilers can perform optimizations that are, arguably, incorrect. This is because many concurrent programs have unrealistic diverging executions that arise when the scheduler starves a thread by never letting it take any more steps—other parts of the program might be waiting on the starving thread to make progress, leading to divergence. The mere existence of such unrealistic diverging executions should not give the compiler license to introduce infinite loops that actually diverge in practice. So we only wish to consider infinite executions exhibited under a *fair* schedule, where no thread is left starving. Fair termination preservation [Liang and Feng 2016] demands that if the optimized program has such an infinite execution, then so does the source program.

Our work establishes fair termination preservation of the example optimizations. In contrast, most variants of CompCert (with the exception of CCAL) do not establish fair termination preservation as part of their correctness proof. (To our knowledge, these compilers are still fair termination-preserving in practice, but that property is not established formally.) We have picked fair termination preservation not because we aim to exploit the assumption of fairness, but because we consider it the more realistic correctness condition for a concurrent compiler.

The remainder of our correctness condition is pretty standard: we demand *refinement* of the final result(s) computed by the program, *i.e.*, whenever the optimized program terminates with a result v, then that should also be a possible result of the original program.[2] Finally, optimizations typically

[2] We are considering languages with non-determinism here, so there might be more than one possible result, and the optimized program might have a subset of the possible results of the original program.

happen locally in some function without knowing the larger program they are a part of. We hence need the optimization to be correct no matter the *context* that this function might be used in.

1.1 Simuliris

In this paper, we introduce *Simuliris*, the first simulation relation to establish fair termination-preserving contextual refinement for concurrent program transformations that can exploit UB.

Iris-style ownership in a coinductive simulation relation. The key idea behind our approach is to leverage the concept of *ownership*: we formulate our simulation relation in a *separation logic* equipped with novel rules for exploiting that data races are undefined behavior. When applied to the motivating example, our proof rules establish that after the unsynchronized load operation `*x`, we gain *ownership* of the underlying memory. This ownership gain is justified due to the undefined behavior of data races. We can hold on to this ownership throughout the entire loop, which lets us prove that each time the original program accesses `*x`, the same result will be returned.

The basic idea of combining separation logic and a simulation relation has been explored before, but prior work has mostly focused on verifying that an *implementation* of an abstract data type implements a *specification*. As such, this line of work lacks the ability to exploit undefined behavior in the way we require for our example (as reflected in Figure 1, and discussed further in §8).

Several of these prior publications explore the use of *Iris* [Jung et al. 2015, 2018b], a framework for defining concurrent separation logics with a flexible form of "ghost ownership", for the purpose of refinement proofs. However, Iris's use of *step-indexing* [Appel and McAllester 2001] means that Iris-based approaches like ReLoC [Frumin et al. 2018] do not support reasoning about liveness properties such as termination preservation. This limitation can be bent to some extent (see §8), but those approaches have not been shown to apply to concurrent programs with unbounded non-determinism.

Simuliris is based on Iris, and as such inherits its flexible notion of ghost state, which forms the foundation for all our ownership reasoning. However, unlike Iris, Simuliris is *not* step-indexed: verification of program transformations requires different tools than general program verification, and we found that the powerful reasoning principles enabled by step-indexing are not required for our task. Simuliris demonstrates how the heart of Iris, its flexible model of ghost state, can be married together with the typical shape of a coinductive simulation to obtain a simulation relation that supports both liveness properties and powerful ownership-based reasoning.

We also inherit the Iris Proof Mode [Krebbers et al. 2017b, 2018], which lets us carry out interactive proofs in the Coq proof assistant for all results presented in this paper (metatheory and examples). The Coq proofs are available in the supplementary material [Gäher et al. 2022].

Fairness and implicit stuttering. One usually rather tedious aspect of termination-preserving simulation relations is *stuttering*. In Simuliris, we manage to completely hide the bookkeeping that is usually associated with stuttering by using a technique we call *implicit stuttering*.

Stuttering is required whenever a lock-step simulation between source and target (*i.e.*, the unoptimized and optimized programs) is insufficient. For example, in the optimization shown above, when the optimized program performs the `*x` before the loop, this does not directly correspond to any step in the source program—so ideally we could just ignore the source when reasoning about this part of the optimized program. However, we have to be careful not to violate termination preservation: if we ignore the source infinitely often, we could end up with a diverging execution in the optimized program even though the source always terminates!

The typical solution to this problem is to add a "stutter counter/metric" that keeps track of how many more steps the target may make before a source step is required. This additional bookkeeping is burdensome and makes it hard to give modular specifications. Instead, we define stuttering *implicitly* (as in [Spies et al. 2021]), with a least fixed-point instead of an explicit decreasing metric. This approach interacts with fairness in non-trivial ways, so we had to develop a new soundness proof to establish that our simulation relation indeed ensures *fair* termination preservation.

Simuliris instances: data races, Stacked Borrows. Like Iris, Simuliris is a language-generic framework. The core simulation relation can be instantiated with multiple different programming languages, providing them with many key proof rules (such as the frame rule from separation logic), a parametric coinduction principle, and an adequacy theorem for fair termination-preserving whole-program refinement "for free". In this paper, we instantiate Simuliris with two different languages.

The first language, SimuLang, is used to demonstrate our ability to perform optimizations based on UB of data races. SimuLang uses the memory model of λ_{Rust} [Jung et al. 2018a], which ensures that programs with data races get stuck. We then perform all of our proofs under the assumption that the source program will *not* get stuck. This permits us to prove correctness of optimizations such as our motivating example by exploiting the absence of data races for unsynchronized accesses: threads fully *own* locations that they perform non-atomic accesses on, hiding the complexity of reasoning about traces of memory events. This layer of abstraction keeps the formal argument fairly close to the intuitive idea of why such an optimization is correct. The language-independent Simuliris infrastructure (in particular framing and coinduction) makes it easy to move from proving individual reorderings to loop hoisting optimizations such as our motivating example.

Our second language is Stacked Borrows [Jung et al. 2020], a recently proposed aliasing model for Rust that supports strong intraprocedural optimizations based on alias information derived from the Rust type system. Originally, correctness of these optimizations was proven in a coinductive ownership-based simulation relation not unlike the model of Simuliris—but without support for concurrency, implicit stuttering, general Iris-style ghost state, or even a separation logic to abstract away resource ownership. Using Simuliris, we verify correctness of the same optimizations as the original paper, but for a new *concurrent* version of Stacked Borrows. We also establish correctness of a new loop hoisting optimization based on Stacked Borrows.

Paper structure. The remainder of the paper is structured as follows: We first give a tour of how Simuliris works in general (§2) and how its approach scales to exploiting undefined behavior of data races (§3) in SimuLang. Then, we turn to the technical meat of the framework: we define our notion of fair termination-preserving contextual refinement (§4), we explain the definition of our underlying simulation relation and how it can be used to establish contextual refinement (§5), and we show how the proof rules for exploiting data races are justified (§6). Finally, we briefly explain how we applied Simuliris to Stacked Borrows (§7), before we conclude with related work (§8) and an overview of limitations and future work (§9).

2 SIMULIRIS BY EXAMPLE: THE BASICS

In this section, we introduce Simuliris's core reasoning principles and show how ownership reasoning helps us to prove program optimizations. We start with a very brief explanation of our setup, and then explain the key rules of Simuliris with a series of simple examples. We illustrate how to use local ownership (§2.1), how to interface with unknown code (§2.2), and how to exploit undefined behavior in the original program (§2.3). We then conclude with an optimization involving loops (§2.4), illustrating that our ownership reasoning nicely composes with coinduction.

At the core of Simuliris lies a coinductive simulation relation defined in separation logic. We write this simulation relation in the style of Relational Hoare Logic [Benton 2004]:

$$\{P\}\ e_t \preceq e_s\ \{v_t, v_s.\ \Phi\ v_t\ v_s\}$$

$$\begin{aligned}
\mathit{Expr} \ni e &::= v \mid x \mid \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \mid e_1 + e_2 \mid \mathsf{call}\ e_1\ e_2 \mid \mathsf{fork}\{e\} \\
&\mid \mathsf{ref}(e) \mid\ !e \mid e_1 \leftarrow e_2 \mid \mathsf{free}(e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{od} \mid \ldots \\
\mathit{Val} \ni v &::= z : \mathbb{Z} \mid b : \mathbb{B} \mid \ell : \mathit{Loc} \mid f : \mathit{FnName} \mid () \mid \ldots \\
\mathit{Prog} \ni \rho &::= (f\ x \triangleq e), \rho \mid \emptyset \\
\mathit{Ectx} \ni K &::= \bullet \mid \mathsf{let}\ x := K\ \mathsf{in}\ e_2 \mid e_1 + K \mid K + v \mid \mathsf{call}\ e_1\ K \mid \mathsf{call}\ K\ v \\
&\mid \mathsf{ref}(K) \mid\ !K \mid e_1 \leftarrow K \mid K \leftarrow v \mid \mathsf{free}(K) \mid \mathsf{if}\ K\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \ldots
\end{aligned}$$

Fig. 2. Excerpt of the grammar of SimuLang.

The idea of these quadruples is that under precondition P, the source expression e_s simulates the target expression e_t such that both terminate in values related by postcondition Φ, or both diverge (as we show below, Φ and P are separation logic assertions that can assert facts about, *e.g.*, the source and target heaps).

Here, e_t and e_s are terms in *SimuLang*, our main example language. An excerpt of the grammar of SimuLang is shown in Figure 2 (omitting some standard features such as products and sums). SimuLang is a simple expression-based language with ML-style references, fork-based concurrency, and function pointers. Evaluation order proceeds right-to-left, as determined by the definition of evaluation contexts K. Further details can be found in our appendix [Gäher et al. 2022, §2].

SimuLang does *not* have λ-terms; one can think of it as a "post-closure-conversion" language. Correspondingly, function pointers simply consist of the name of a function. A whole SimuLang program ρ is a list of mutually recursive function declarations $f\ x \triangleq e$ (we implicitly assume that no function name is declared more than once). Any call to this function $\mathsf{call}\ f\ v$ reduces to e with the call argument v substituted for x. Local variables inside a function are bound via let. These variables are immutable; to model mutable stack-allocated variables, we use $\mathsf{ref}(\cdot)$. In other words, we do not distinguish between stack-allocated and heap-allocated variables (following prior languages designed for the verification of Rust or C code [Jung et al. 2018a; Sammler et al. 2021]). We use $e_1; e_2$ as sugar for $\mathsf{let}\ \_ := e_1\ \mathsf{in}\ e_2$. SimuLang features while loops: the term $\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{od}$ reduces to $\mathsf{if}\ e_1\ \mathsf{then}\ (e_2; \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{od})\ \mathsf{else}\ ()$ (and there is no evaluation context for loops).

In the following, when writing concrete programs, we will use sans-serif font for typesetting concrete names of program variables or functions (represented as strings in the formal Coq development), while using typical *italic* font for logical variables.

2.1 Optimizations on Local Memory Locations

To explain how we use ownership reasoning for compiler optimizations, let us consider a simple example: removing a load from a local (unescaped) memory location. In SimuLang, the expression $\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\ !\mathsf{y}$ allocates a fresh location and then reads from it[3]. We would like to optimize the load of $!\mathsf{y}$ away and directly return 42. This optimization is correct because the value stored in y cannot change between the allocation and the load: no other thread can know about y and the memory location it references. In other words, it is as if we exclusively *own* this location. In the following, we will show how we can make this argument formal with Simuliris.

We verify the above optimization by showing the following quadruple:

$$\{\mathsf{True}\}\ \mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\ 42 \preceq \mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\ !\mathsf{y}\ \{v_t, v_s.\ v_t = v_s\}$$

[3] In order to simplify the presentation, many examples in this paper leak memory by omitting deallocations. Of course, our framework supports the same optimizations in the presence of proper deallocation.

Language-independent rules:

SOURCE-FOCUS
$$\frac{\{P\}\ e_s\ \{v_s.\ \Psi\ v_s\}^{\mathsf{src}} \qquad \forall v_s.\ \{\Psi\ v_s\}\ e_t \preceq K_s[\,v_s\,]\ \{\Phi\}}{\{P\}\ e_t \preceq K_s[\,e_s\,]\ \{\Phi\}}$$

SIM-FRAME
$$\frac{\{P\}\ e_t \preceq e_s\ \{\Phi\}}{\{P * R\}\ e_t \preceq e_s\ \{v_t, v_s.\ \Phi\ v_t\ v_s * R\}}$$

SIM-CALL
$$\{\mathcal{V}\ v_t\ v_s\}\ \mathsf{call}\ \mathsf{f}\ v_t \preceq \mathsf{call}\ \mathsf{f}\ v_s\ \{v_t', v_s'.\ \mathcal{V}\ v_t'\ v_s'\}$$

SIM-BIND
$$\frac{\{P\}\ e_t \preceq e_s\ \{v_t, v_s.\ \Psi\ v_t\ v_s\} \qquad \forall v_t, v_s.\ \{\Psi\ v_t\ v_s\}\ K_t[\,v_t\,] \preceq K_s[\,v_s\,]\ \{\Phi\}}{\{P\}\ K_t[\,e_t\,] \preceq K_s[\,e_s\,]\ \{\Phi\}}$$

SIM-VALUE
$$\{\Phi\ v_t\ v_s\}\ v_t \preceq v_s\ \{\Phi\}$$

SimuLang rules:

SOURCE-ALLOC
$$\{\mathsf{True}\}\ \mathsf{ref}(v_s)\ \{v_s'.\ \exists \ell_s.\ v_s' = \ell_s * \ell_s \mapsto^{\mathsf{src}} v_s\}^{\mathsf{src}}$$

TARGET-ALLOC
$$\{\mathsf{True}\}\ \mathsf{ref}(v_t)\ \{v_t'.\ \exists \ell_t.\ v_t' = \ell_t * \ell_t \mapsto^{\mathsf{tgt}} v_t\}^{\mathsf{tgt}}$$

SOURCE-PURE
$$\frac{e_s \to^*_{\mathsf{pure}} e_s' \qquad \{P\}\ e_t \preceq e_s'\ \{\Phi\}}{\{P\}\ e_t \preceq e_s\ \{\Phi\}}$$

TARGET-PURE
$$\frac{e_t \to^*_{\mathsf{pure}} e_t' \qquad \{P\}\ e_t' \preceq e_s\ \{\Phi\}}{\{P\}\ e_t \preceq e_s\ \{\Phi\}}$$

SOURCE-LOAD
$$\{\ell_s \mapsto^{\mathsf{src}} v_s\}\ !\ell_s\ \{v_s'.\ v_s' = v_s * \ell_s \mapsto^{\mathsf{src}} v_s\}^{\mathsf{src}}$$

TARGET-LOAD
$$\{\ell_t \mapsto^{\mathsf{tgt}} v_t\}\ !\ell_t\ \{v_t'.\ v_t' = v_t * \ell_t \mapsto^{\mathsf{tgt}} v_t\}^{\mathsf{tgt}}$$

SOURCE-STORE
$$\{\ell_s \mapsto^{\mathsf{src}} \_\}\ \ell_s \leftarrow v_s\ \{v_s'.\ v_s' = () * \ell_s \mapsto^{\mathsf{src}} v_s\}^{\mathsf{src}}$$

TARGET-STORE
$$\{\ell_t \mapsto^{\mathsf{tgt}} \_\}\ \ell_t \leftarrow v_t\ \{v_t'.\ v_t' = () * \ell_t \mapsto^{\mathsf{tgt}} v_t\}^{\mathsf{tgt}}$$

Fig. 3. Core Simuliris and SimuLang rules.

Under the trivial precondition, we can execute the optimized (left, target) and the original (right, source) program, resulting in the same value.

To prove such quadruples, our simulation relation allows us to "focus" on the execution of source and target individually by switching to special source and target *triples*, with rules like SOURCE-FOCUS (see Figure 3; the target rule is symmetric). As we will see below, most of the rules for these triples are symmetric and reminiscent of a unary separation logic. With these triples, we can focus on a subexpression of either the source or the target. The focusing rules combine sequencing and *implicit* stuttering: we can focus on an expression in evaluation position (*i.e.*, which is contained in an evaluation context K) on one side and then show that that expression terminates in a value[4] satisfying Ψ (it must not diverge). Meanwhile, the other side of the program "stutters", not making any progress. Afterwards, the postcondition Ψ can be used to continue with the rest of the simulation. (The reader might wonder at this point how stuttering without further side-conditions does not break termination preservation. We will explain this in §5.)

With the focusing triples in hand, we can start the proof. First, we focus on $\mathsf{ref}(42)$ in the source with SOURCE-FOCUS to allocate a memory location, using SOURCE-ALLOC. As mentioned above, we obtain *local* ownership of this memory location—no other parts of the program can know about it.[5] In ordinary separation logic, this notion of ownership can be expressed with the points-to connective $\ell \mapsto v$, stating both the knowledge that ℓ contains value v and exclusive ownership of

[4]As we will see in §2.4, our full system also supports leaving the focus early with an expression-based postcondition.
[5]In addition, allocation provides us with ownership of assertions about the size of allocations, which are relevant for deallocation and pointer arithmetic. For simplicity, we omit this detail and refer to the supplementary material.

location ℓ. As Simuliris is a *relational* separation logic [Yang 2007], it has *two* points-to connectives: $\ell \mapsto^{\text{tgt}} v$ for locations in the target, and $\ell \mapsto^{\text{src}} v$ for locations in the source. SOURCE-ALLOC picks a fresh location ℓ_s and provides the corresponding local ownership of this source location.

Having reduced the source expression, we are left with the goal $\left\{\ell_s \mapsto^{\text{src}} 42\right\}$ $\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\ 42 \preceq \mathsf{let}\ \mathsf{y} := \ell_s\ \mathsf{in}\ !\mathsf{y}\ \{v_t, v_s.\, v_t = v_s\}$. To allocate the reference in the target, we follow the same steps as in the source: we first apply the target focusing rule (which works exactly like source focusing), and then use TARGET-ALLOC. (Ownership of $\ell_s \mapsto^{\text{src}} 42$ is maintained using the standard frame rule, which we only show for the relational case: SIM-FRAME.) To then execute $\mathsf{ref}(42)$, we use TARGET-ALLOC. We are left to prove $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}$ $\mathsf{let}\ y := \ell_t\ \mathsf{in}\ 42 \preceq \mathsf{let}\ y := \ell_s\ \mathsf{in}\ !\ell_s\ \{v_t, v_s.\, v_t = v_s\}$.

Next, we reduce the let-expression in source and target with the pure execution rules SOURCE-PURE and TARGET-PURE. We now end up with the remaining goal $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}$ $42 \preceq\ !\ell_s\ \{v_t, v_s.v_t = v_s\}$. At this point, we make a source execution step that is not mirrored in the target (again using our focusing triples): we load 42 from ℓ_s. To do so, it is crucial that we have $\ell_s \mapsto^{\text{src}} 42$ in our precondition, since it means we exclusively *own* the location ℓ_s and hence there was no interference by other threads. Specifically, we apply the rule SOURCE-LOAD.

We are left with $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}$ $42 \preceq 42\ \{v_t, v_s.v_t = v_s\}$. Using SIM-VALUE, establishing the postcondition is trivial since we only have to show the equality of 42 and 42.

2.2 Interfacing with External Code

The previous optimization exploits ownership to ensure that a memory location cannot be modified between allocation and a load. However, ownership reasoning carries a lot further: we can also use it to argue that external, unknown code in the current thread cannot modify the location.

Function calls. Consider the following example where function f (which we assume to know nothing about[6]) is called between an allocation and a load:

$$\{\mathsf{True}\}\quad \begin{array}{l}\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\\ \mathsf{call}\ \mathsf{f}\ 23;\ 42\end{array} \quad\preceq\quad \begin{array}{l}\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\\ \mathsf{call}\ \mathsf{f}\ 23;\ !\mathsf{y}\end{array} \quad \{v_t, v_s.\, v_t = v_s\}$$

The start of the proof proceeds as before: we allocate the locations in source and target, so the program variable y is replaced by some location ℓ_s in the source and (potentially different) location ℓ_t in the target. Next, we focus on the call to f with the sequencing rule SIM-BIND, which enables us to first show a simulation of two subexpressions in evaluation position before considering the surrounding context.

Thus, we are left with $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}$ $\mathsf{call}\ \mathsf{f}\ 23 \preceq \mathsf{call}\ \mathsf{f}\ 23$ $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}$ (the return value does not matter in this case). We now exploit that our simulation relation allows us to skip over calls to the *same* function f in source and target, provided that we call it with "sufficiently similar" arguments v_s and v_t. We get to assume that the function behaves the same on both sides and returns "sufficiently similar" values. Formally, this is captured by SIM-CALL, where the *value relation* $\mathcal{V}\ v_t\ v_s$ captures the notion of being "similar". As shown in Figure 4, for simple cases like integers, the value relation just requires syntactic equality, while for composite values like pairs it is lifted componentwise. The more complicated case of locations will be discussed below.

Like all our proof rules, SIM-CALL is compatible with framing, so we can just frame ownership of ℓ_t and ℓ_s around the call. In this case, however, that is a remarkably powerful reasoning principle! It reflects the idea that local locations are not accessible to unknown function calls, and thus not

[6] Simuliris also has rules for the (less interesting) case of functions for which we know the source code. In that case, we can just step into the function and reason as if it was inlined. We can also prove and apply lemmas about individual functions in source or target with non-trivial postconditions, using standard separation logic reasoning.

$$\mathcal{V}\ z_t\ z_s \triangleq z_t = z_s \qquad \text{(VALUE-INT)} \qquad \mathcal{V}\ \ell_t\ \ell_s \triangleq \ell_t \leftrightarrow_h \ell_s \qquad \text{(VALUE-LOC)}$$
$$\mathcal{V}\ b_t\ b_s \triangleq b_t = b_s \qquad \text{(VALUE-BOOL)} \qquad \mathcal{V}\ f_s\ f_t \triangleq f_s = f_t \qquad \text{(VALUE-FNPTR)}$$
$$\mathcal{V}\ (v_t, w_t)\ (v_s, w_s) \triangleq \mathcal{V}\ v_t\ v_s * \mathcal{V}\ w_t\ w_s \qquad \text{(VALUE-PAIR)}$$

Fig. 4. Excerpt of the value relation. Values of different kinds (*e.g.*, integers and locations) are not related.

affected by those calls. Separation logic lets us express this at a very high level of abstraction, making this kind of reasoning effortless even when doing mechanized interactive proofs in Coq.

After using SIM-CALL and SIM-FRAME, we are left with the goal $\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}\ 42 \preceq\ !\ell_s\ \{v_t, v_s.\ v_t = v_s\}$. We can now complete the proof in the same way as before.

But let us backtrack for a moment: how can it be sound to just skip over function calls for arbitrary functions f? Intuitively, a compiler should be able to reason locally about function bodies, without making strong assumptions about the behavior of unknown functions the program might be linked against. The key to enabling this is to make our simulation relation *open*, allowing to skip calls to the *same* function in the source and target. But of course, we have to work for this: to enable SIM-CALL, the function we skip over needs to respect ownership (which enabled us to frame local ownership around the call)! The top-level adequacy proof (§5) will thus assume that *all* functions in the program satisfy the simulation relation, tying a big mutually recursive knot.

Passing pointers to functions. Above, we have postponed discussion of the value relation for locations. To illustrate what happens if we pass a location to a function, we consider the following example: we optimize away a load from a pointer that we pass to a function afterwards.

$$\{\mathsf{True}\}\ \begin{array}{l}\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\\ \mathsf{let}\ \mathsf{z} := 42\ \mathsf{in}\\ \mathsf{call}\ \mathsf{f}\ \mathsf{y};\ !\mathsf{y} + \mathsf{z}\end{array} \preceq \begin{array}{l}\mathsf{let}\ \mathsf{y} := \mathsf{ref}(42)\ \mathsf{in}\\ \mathsf{let}\ \mathsf{z} := !\mathsf{y}\ \mathsf{in}\\ \mathsf{call}\ \mathsf{f}\ \mathsf{y};\ !\mathsf{y} + \mathsf{z}\end{array}\ \{v_t, v_s.\ v_t = v_s\}$$

As before, we start by (1) allocating the pointers in source and target and (2) loading from the pointer in the source, leveraging local ownership. We end up with the goal:

$$\left\{\ell_t \mapsto^{\text{tgt}} 42 * \ell_s \mapsto^{\text{src}} 42\right\}\ \mathsf{call}\ \mathsf{f}\ \ell_t; !\ell_t + 42 \preceq \mathsf{call}\ \mathsf{f}\ \ell_s; !\ell_s + 42\ \{v_t, v_s.\ \mathcal{V}\ v_t\ v_s\}$$

When passing pointers to unknown functions, we have to ensure that they in turn point to "sufficiently similar" values, as those can be observed by the callee. Moreover, the location must not be exclusively owned, since the callee could mutate memory and thus violate the ownership discipline. Both of these aspects are captured by the assertion $\ell_t \leftrightarrow_h \ell_s$ which states that target location ℓ_t and source location ℓ_s form an *escaped* pair of locations. We can convert locally owned locations to escaped locations by giving up ownership with the rule LOC-ESCAPE:

LOC-ESCAPE
$$\frac{\{\ell_t \leftrightarrow_h \ell_s * P\}\ e_t \preceq e_s\ \{\Phi\}}{\left\{\ell_t \mapsto^{\text{tgt}} v_t * \ell_s \mapsto^{\text{src}} v_s * \mathcal{V}\ v_t\ v_s * P\right\}\ e_t \preceq e_s\ \{\Phi\}}$$

SIM-LOAD-ESCAPED
$$\{\ell_t \leftrightarrow_h \ell_s\}\ !\ell_t \preceq\ !\ell_s\ \{v_t, v_s.\ \mathcal{V}\ v_t\ v_s\}$$

The assertion $\ell_t \leftrightarrow_h \ell_s$ represents knowledge, not exclusive ownership, and as such it is freely duplicable. We can thus retain it beyond the function call and are left with the remaining goal $\{\ell_t \leftrightarrow_h \ell_s\}\ !\ell_t + 42 \preceq\ !\ell_s + 42\ \{v_t, v_s.\ v_t = v_s\}$. Now, for the last step, we load both locations simultaneously in with SIM-LOAD-ESCAPED (see above). Since the loaded values are integers related by $\mathcal{V}$, they must therefore be equal, establishing the postcondition.[7] Note that, as we have given up ownership of ℓ_s to escape it, we cannot be sure that ℓ_s still stores 42, and thus cannot optimize away the load

[7] Here we implicitly exploit that addition would go wrong if these were not integers; this technique is described more precisely in the next subsection.

$$\frac{\text{SIM-SRC-SAFE}\quad e_s \rightsquigarrow Q \qquad \{Q * P\}\ e_t \preceq e_s\ \{\Phi\}}{\{P\}\ e_t \preceq e_s\ \{\Phi\}}$$

$$\text{SAFE-DIV-INT-NONZERO}\quad v/w \rightsquigarrow \exists z_1, z_2 : \mathbb{Z}.\ v = z_1 \land w = z_2 \land z_2 \neq 0$$

Fig. 5. Subset of the rules for exploiting source UB.

$$\{\mathsf{True}\}\quad \begin{array}{l}\mathsf{let\ x := ref(0)\ in\ let\ r := !x\ in}\\ \mathsf{while\ (call\ f\ r)\ do\ call\ g\ r\ od}\end{array} \quad\preceq\quad \begin{array}{l}\mathsf{let\ x := ref(0)\ in}\\ \mathsf{while\ (call\ f\ !x)\ do\ call\ g\ !x\ od}\end{array}\quad \{v_t, v_s.\ \mathsf{True}\}$$

Fig. 6. Loop-invariant code motion for a load.

after the call as we did before. This is for a good reason: the function we have passed the location to could just have written a different value to it.

2.3 Exploiting Undefined Behavior

So far, we have seen how our simulation relation allows us to prove optimizations that rely on ownership reasoning. We now introduce another core component of Simuliris: the ability to exploit *undefined behavior* (UB) in the source program. We will later leverage this feature for reasoning about data races (§3), but for now we focus on a simple example:

$$\{\mathsf{True}\}\quad \begin{array}{l}\mathsf{let\ (x,y) := call\ f\ ()\ in}\\ \mathsf{let\ z := x/y\ in}\\ \mathsf{call\ g\ z}\end{array} \quad\preceq\quad \begin{array}{l}\mathsf{let\ (x,y) := call\ f\ ()\ in}\\ \mathsf{let\ z := x/y\ in}\\ \mathsf{if\ y \neq 0\ then\ call\ g\ z\ else\ call\ h\ z}\end{array}\quad \{v_t, v_s.\ \mathsf{True}\}$$

Here, we optimize away the conditional and, instead, always pick the "then" branch in the optimized program. The reason why this optimization is correct is that, in SimuLang (similar to C), division by 0 is *UB*. As a compiler, we may assume that the original program (the source program) does not have UB. It then follows that in all relevant (UB-free) executions, $y \neq 0$.

In Simuliris, this kind of reasoning is exposed through the judgment $e_s \rightsquigarrow Q$, which expresses that Q follows from the assumption that e_s does not have UB.[8] The relevant rules for our example are given in Figure 5. In the case of division, we can use them to prove the quadruple $\{\mathsf{True}\}\ v/w \preceq v/w\ \{v_t, v_s.\ v_t = v_s * w \neq 0\}$. This quadruple not only ensures that the resulting values are the same, but also that the value w, the divisor, is not 0. Importantly, the fact that w is not 0 is not a precondition that we need to provide—it appears in our postcondition, so it is something we learn! We can then use $w \neq 0$ afterwards to prove that the conditional will always pick the left branch.

2.4 Reasoning About Loops

All the examples we have considered so far did not contain any loops. We now show that the above reasoning principles extend naturally to programs with loops.

To this end, we use the example of *hoisting a load out of a loop*. More precisely, as depicted in Figure 6, we avoid repeated loads from the pointer x inside a while-loop and, instead, load only once from x before the loop starts. The condition and body of the loop are defined by arbitrary functions f and g, demonstrating that the reasoning works without any knowledge of what the loop is actually doing or whether it even terminates. (One could trivially optimize this code further, but we focus on loop reasoning here.)

[8] The assumed UB-freedom states a property not only about the next reduction step, but about any execution of e_s.

Intuitively, this optimization is correct because we have local ownership of x and, additionally, x is not modified in the loop. We will now make this argument formal in Simuliris. The initial allocation of x and also the new load in the target are covered using the principles we have seen before. What is more interesting is what we do about the loop: we will use a *coinductive* argument to justify this optimization. Specifically, we will use the following coinduction principle for while-loops:

WHILE-COIND

$$\frac{\{I\}\ c_t \preceq c_s\ \{v_t, v_s.\ \mathcal{V}\ v_t\ v_s * ((v_s = \textbf{true} * J) \lor (v_s \neq \textbf{true} * Q))\} \qquad \{J\}\ e_t \preceq e_s\ \{I\}}{\{I\}\ \text{while}\ c_t\ \text{do}\ e_t\ \text{od} \preceq \text{while}\ c_s\ \text{do}\ e_s\ \text{od}\ \{Q\}}$$

The idea of this rule is the following: when we apply it, we may pick a loop invariant I. We then first show that given the invariant I, either the loop conditions c_t and c_s (ordinary expressions) both evaluate to **true** and a proposition J holds, or they both do not evaluate to **true** and the postcondition Q of the loop holds. (The postcondition is not simply the negation of the loop condition, because we also need to transfer ownership.) Second, we need to show that if we execute the loop bodies e_t and e_s assuming J, then the loop invariant I is preserved.

Applying this rule is the key step of verifying the example in Figure 6. At the point where we have to reason about the loop, our current precondition is $\ell_t \mapsto^{\text{tgt}} 0 * \ell_s \mapsto^{\text{src}} 0$ where ℓ_t and ℓ_s are the source and target values of x. We proceed by applying first SIM-BIND to focus on the loops and then WHILE-COIND to reason about the loop. As the invariant, we pick $I \triangleq (\ell_t \mapsto^{\text{tgt}} 0 * \ell_s \mapsto^{\text{src}} 0)$ (our precondition). It is straightforward to show both premises of the rule WHILE-COIND.

Parameterized coinduction. The rule WHILE-COIND is already sufficient to verify the previous example. However, there are interesting loops for which WHILE-COIND is not strong enough. For example, what if I only holds after every other iteration (*e.g.*, it asserts that some loop counter is even)? To also cover such cases, our simulation supports parameterized coinduction [Hur et al. 2013]. Whereas regular coinduction requires re-establishing the invariant after each iteration, parameterized coinduction says that we can take any number of rounds through the loop and then re-establish the invariant whenever we are "back at the beginning":

WHILE-PACO

$$\frac{\begin{array}{c} W_t = \text{while}\ c_t\ \text{do}\ e_t\ \text{od} \qquad W_s = \text{while}\ c_s\ \text{do}\ e_s\ \text{od} \\ \{I\}\ \text{if}\ c_t\ \text{then}\ e_t; W_t\ \text{else}\ () \preceq \text{if}\ c_s\ \text{then}\ e_s; W_s\ \text{else}\ ()\ \left\{e'_t, e'_s.\ \Phi\ e'_t\ e'_s \lor (e'_t = W_t * e'_s = W_s * I)\right\} \end{array}}{\{I\}\ W_t \preceq W_s\ \left\{e'_t, e'_s.\ \Phi\ e'_t\ e'_s\right\}}$$

In WHILE-PACO, the while loops are first reduced for one step, entering the first loop iteration (as a guard for the coinduction). Thereafter, we can finish the proof—at any point—by showing either (1) the postcondition Φ or (2) that the current target and source expressions are the W_t and W_s that we started with *and* the invariant I holds again. The latter condition is expressed by "parameterizing" the new proof goal through an additional disjunct in the postcondition. To enable this, we generalize our simulation relation (and the source/target triples) from a postcondition on values to a *postcondition on expressions*, such that one can finish the proof at any time if the current expressions satisfy the postcondition: $\{\Phi\ e_t\ e_s\}\ e_t \preceq e_s\ \left\{e'_t, e'_s.\ \Phi\ e'_t\ e'_s\right\}$. The reasoning principles that we have seen so far (*e.g.*, SIM-BIND, SOURCE-FOCUS, and SIM-FRAME) all generalize to a simulation relation with expression postconditions and the generalized rules can be found in the supplementary material [Gäher et al. 2022]. The simulation relation with a postcondition on values can be derived from the more general form, by requiring in the postcondition that the expressions have terminated.

Adding and removing finite loops. The rules WHILE-COIND and WHILE-PACO can be used to reason about situations where a loop exists in both the source and target programs, and we need to "match up" their loop iterations to argue that we are not introducing an infinite loop or making

$$\{z_1 \geq 0\} \quad z_1 * z_2 \quad \preceq \quad \begin{array}{l} \text{let } \mathsf{n} := \text{ref}(z_1) \text{ in let } \mathsf{x} := \text{ref}(0) \text{ in} \\ \text{while } 0 < {!\mathsf{n}} \text{ do } \mathsf{x} \leftarrow {!\mathsf{x}} + z_2; \mathsf{n} \leftarrow {!\mathsf{n}} - 1 \text{ od}; \\ {!\mathsf{x}} \end{array} \quad \{v_t, v_s.\, v_t = v_s\}$$

Fig. 7. Optimization for replacing repeated addition with primitive multiplication.

previously unreachable code reachable. Simuliris also supports reasoning principles for loops that occur only on one side of the simulation, provided that we can ensure that the loop only takes a finite number of iterations. For instance, consider the simulation shown in Figure 7: here we replace the multiplication of two integers z_1 and z_2 through repeated addition with a single primitive multiplication instruction.[9] Our proof exploits the fact that the loop is terminating. After initializing n and x, we perform induction on the value stored in n (initially z_1). We unfold the source loop for one step, using the reduction rule $\text{while } c \text{ do } e \text{ od} \rightarrow_{\text{pure}} \text{if } c \text{ then } (e; \text{while } c \text{ do } e \text{ od}) \text{ else } ()$ and SOURCE-PURE. If the loop condition is true, the value stored in n is decreased before the next iteration, and so we can use the inductive hypothesis.

The reason this example works is that we embed standard reasoning principles from the meta level into Simuliris, including support for reasoning about integers and induction. More precisely, the mechanization of Simuliris is a shallow embedding into Coq's logic. As a consequence, Coq's existing types and the reasoning principles for their elements become readily available in Simuliris. In particular, Simuliris inherits from Coq standard data types (*e.g.*, integers, finite maps, lists, and strings), recursive functions, and induction principles for inductive types.

3 SIMULIRIS BY EXAMPLE: EXPLOITING NON-ATOMIC ACCESSES

Now that we have seen the basics of Simuliris, let us look at a class of challenging optimizations where the ownership reasoning provided by Simuliris enables an elegant verification technique: optimizations around non-atomic accesses as found in languages like C. First, we introduce the basic principles of non-atomic accesses (§3.1), then we introduce proof rules for verifying optimizations around such accesses (§3.2), and finally we verify the optimization from the introduction (§3.3).

3.1 Basics of Non-atomics

Previously, we have seen how ownership enables a wide variety of optimizations (§2) for *local* memory locations that have not been leaked to other, unknown code yet. However, what about *escaped* locations? At first, it may seem impossible to optimize accesses to escaped locations since other threads might interfere and invalidate the optimization by overwriting the memory in a racy way. This is reflected in the fact that one has to give up ownership of the location when escaping it.

However, compilers have another trick up their sleeve to enable optimizations even for escaped locations: it is based on the observation that only very few accesses to escaped locations actually race with other threads (which would invalidate optimizations), so the programmer should mark such accesses as *atomic* accesses. Atomic accesses can be used to provide synchronization between different threads and are for example used for the implementation of synchronization primitives like locks. For all other accesses, called *non-atomic* accesses, the programmer guarantees that they will not race with other threads, and thus they can be optimized more heavily. Technically, this works by assigning *undefined behavior* (UB) to executions with races on non-atomic accesses, and thus such executions do not have to be considered when verifying an optimization.

[9] We can also prove the inverse direction in exactly the same way. This is a good example of the power of implicit stuttering: we are adding an unbounded number of program steps, and the proof implicitly shows that the program still terminates.

In this paper, we follow the definition of non-atomic accesses used by RustBelt [Jung et al. 2018a] and RefinedC [Sammler et al. 2021]: first, we distinguish between non-atomic accesses ($\mathsf{x} \leftarrow \mathsf{y}$ and $!\mathsf{x}$) and sequentially consistent atomic accesses ($\mathsf{x} \leftarrow^{sc} \mathsf{y}$ and $!^{sc}\mathsf{x}$). Second, we extend the operational semantics with a data-race detector that raises UB if two threads access the same location at the same time, where at least one access is a store and at least one access is non-atomic.[10]

As an example of optimizations enabled by the addition of data-races, consider the following transformation that replaces a load from x with the value of a previous (non-atomic) store (assuming that the omitted code in ... does not write to x or perform atomic stores):

$$\mathsf{x} \leftarrow 42; \ldots; !\mathsf{x} \xrightarrow{\text{optimized to}} \mathsf{x} \leftarrow 42; \ldots; 42$$

The optimization can be justified as follows: if there is no store to x by another thread between the first and the second statement, x still contains the value 42 (because the code in ... does not write to x) and the optimization is correct. Otherwise, the programs behave differently, but the optimization is still correct because we can show that this case is impossible for well-defined executions (without UB). We do this as follows: we know that there is another thread that performs a store to x. If this concurrent store occurs at the same time as our store to x, the race detector raises UB and we are done. However, what if in the execution we are considering, the other store is delayed, so that the two stores do not occur at the same time? The race detector would miss that data race. But in this case we can construct an *alternative* thread interleaving where the concurrent stores *do* happen at the same time, and in this interleaving the race detector raises UB—this reordering is possible because there is no atomic store in the omitted code. Since we can assume that *no* interleaving raises UB, we are done.

3.2 Justifying Optimizations of Non-atomic Accesses via Ownership

We have seen that the correctness of optimizations of non-atomic accesses relies on some subtle reasoning about UB and alternative thread interleavings. This section shows how we build a verification technique for SimuLang that abstracts over these details and only exposes simple yet powerful separation logic rules for justifying optimizations around non-atomic accesses. The key insight is that we can use the presence of a non-atomic access to gain *ownership* of exposed locations, which in turn enables many of the proof rules seen before for local locations to apply. For the example above, we know that no other thread could access the location x without raising UB, giving us exclusive ownership of x. With this ownership, we can prove that the load of x returns 42. To make this argument formal, we prove the following quadruple:

$$\begin{gathered}\{x_t \leftrightarrow_h x_s * y_t \leftrightarrow_h y_s * \mathsf{exploit}\ \pi\ \emptyset\} \\ x_t \leftarrow 42; !^{sc} y_t;\ 42 \qquad \preceq_\pi \qquad x_s \leftarrow 42; !^{sc} y_s;\ !x_s \\ \{v_t, v_s.\ v_t = v_s * \mathsf{exploit}\ \pi\ \emptyset\}\end{gathered}$$

Note that this quadruple uses an atomic load from an escaped location y as a concrete but interesting instance of the code between the store and the load.[11] We also need a new logical assertion, $\mathsf{exploit}\ \pi\ \emptyset$, that we will explain during the proof.

Rules for exploiting UB of non-atomic accesses. The first step of verifying the example is to gain ownership of x_s and x_t by "exploiting" the non-atomic store to x_s. We do this via the rule EXPLOIT-STORE in Figure 8: it lets us acquire ownership of a source location ℓ_s (here $x_s \mapsto^{\text{src}} v_s$) and related target location ℓ_t (here $x_t \mapsto^{\text{tgt}} v_t$) given a reachable non-atomic store in the source

[10] A detailed description of the data-race detector can be found in Jung [2020, §9.2] and Gäher et al. [2022, §2].

[11] We have also verified this example for arbitrary read-only code between the store and the load, but this requires some further machinery that we omit for the purpose of presentation.

EXPLOIT-STORE

$$\frac{C(\ell_s) = \bot \qquad e_s \to_?^* K[\, \ell_s \leftarrow v_0 \,] \qquad \forall v_t, v_s.\ \left\{\ell_s \mapsto^{\mathrm{src}} v_s * \ell_t \mapsto^{\mathrm{tgt}} v_t * \mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ (C, \ell_s \mapsto \mathsf{Wr}) * P\right\} e_t \preceq_\pi e_s \{\Phi\}}{\{\ell_t \leftrightarrow_h \ell_s * \mathsf{exploit}\ \pi\ C * P\}\ e_t \preceq_\pi e_s\ \{\Phi\}}$$

EXPLOIT-LOAD

$$\frac{C(\ell_s) = \bot \qquad e_s \to_?^* K[\, !\ell_s \,] \qquad \forall v_t, v_s, q.\ \left\{\ell_s \overset{q}{\mapsto}{}^{\mathrm{src}} v_s * \ell_t \overset{q}{\mapsto}{}^{\mathrm{tgt}} v_t * \mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ (C, \ell_s \mapsto \mathsf{Rd}(q)) * P\right\} e_t \preceq_\pi e_s \{\Phi\}}{\{\ell_t \leftrightarrow_h \ell_s * \mathsf{exploit}\ \pi\ C * P\}\ e_t \preceq_\pi e_s\ \{\Phi\}}$$

RELEASE-EXPLOIT

$$\frac{C(\ell_s) = c \qquad q = \mathsf{exploit\_frac}(c) \qquad \{\mathsf{exploit}\ \pi\ (C \setminus \ell_s) * P\}\ e_t \preceq_\pi e_s\ \{\Phi\}}{\left\{\ell_s \overset{q}{\mapsto}{}^{\mathrm{src}} v_s * \ell_t \overset{q}{\mapsto}{}^{\mathrm{tgt}} v_t * \mathcal{V}\ v_t\ v_s * \ell_t \leftrightarrow_h \ell_s * \mathsf{exploit}\ \pi\ C * P\right\} e_t \preceq_\pi e_s\ \{\Phi\}}$$

$$\mathsf{exploit\_frac}(\mathsf{Rd}(q)) \triangleq q \qquad\qquad \mathsf{exploit\_frac}(\mathsf{Wr}) \triangleq 1$$

Fig. 8. Rules for exploiting and releasing ownership from non-atomic accesses.

program.[12] These ownership assertions are the same as for local locations, which consequently allows us to use the rules for loads and stores presented in Figure 3 (§2.1) for the verification of our example. However, we need to ensure that EXPLOIT-STORE is not used twice for the same location, as it is not sound to gain exclusive ownership of the same location twice. This is achieved via the assertion exploit π C, which tracks which locations have been exploited by the current thread π via EXPLOIT-STORE. (exploit π C is also relevant for other rules as we will see later.) EXPLOIT-STORE first checks that ℓ_s is currently not already exploited (via the side-condition $C(\ell_s) = \bot$ which holds trivially in our example since $C = \emptyset$). Then it records that one has acquired ownership for ℓ_s via a store (denoted by Wr for "write") in C (resulting in exploit π $(x_s \mapsto \mathsf{Wr})$ in our example). To be able to talk about the "current thread", we also equip the simulation relation with a thread id π.

The rule for exploiting loads EXPLOIT-LOAD is similar to EXPLOIT-STORE except that it does not provide full ownership but only *fractional ownership* [Boyland 2003; Bornat et al. 2005] with some fraction q. This gives us permission to introduce extra loads in the target program, but we cannot introduce extra stores as that would require full ownership. Correspondingly, EXPLOIT-LOAD records that one has acquired the fraction q by adding $\mathsf{Rd}(q)$ (for "read") to exploit π C.

For our example, after using EXPLOIT-STORE and obtaining ownership of $x_t \mapsto^{\mathrm{tgt}} v_t$ and $x_s \mapsto^{\mathrm{src}} v_s$ for some v_t, v_s, we execute the stores in source and target and transform the ownership to $x_t \mapsto^{\mathrm{tgt}} 42$ and $x_s \mapsto^{\mathrm{src}} 42$. Next, we verify the atomic load in both source and target via the rule SIM-LOAD-SC in Figure 9. This rule is similar to SIM-LOAD-ESCAPED except that we need to prove that the location ℓ_s (here y_s) is not in C. (In fact, as we will see in §6, SIM-LOAD-ESCAPED is no longer sound when adding the ability to exploit non-atomic accesses and must be replaced by SIM-LOAD-NA with a similar sidecondition.) We only consider the case $x_s \neq y_s$ where this side-condition holds. (For $x_s = y_s$, we can use the ownership of x_s gained earlier to perform the loads.) For the final load, we use SOURCE-LOAD to prove that the load in the source returns 42, which ensures that both sides return the same value. Finally, we reestablish exploit π $\emptyset$ by giving back $x_t \mapsto^{\mathrm{tgt}} 42$ and $x_s \mapsto^{\mathrm{src}} 42$ via RELEASE-EXPLOIT, which lets us end the exploitation of a location.

[12] The notion of reachability $e \to_?^* e'$ used by EXPLOIT-STORE and EXPLOIT-LOAD will be explained in §3.3 where it is used in a non-trivial way.

SIM-LOAD-SC
$$\{C(\ell_s) = \bot * \ell_t \leftrightarrow_h \ell_s * \mathsf{exploit}\ \pi\ C\}\ !^{sc}\ell_t \preceq_\pi\ !^{sc}\ell_s\ \{v_t, v_s.\ \mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ C\}$$

SIM-LOAD-NA
$$\{C(\ell_s) = \bot * \ell_t \leftrightarrow_h \ell_s * \mathsf{exploit}\ \pi\ C\}\ !\ell_t \preceq_\pi\ !\ell_s\ \{v_t, v_s.\ \mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ C\}$$

SIM-STORE-SC
$$\{\ell_t \leftrightarrow_h \ell_s * \mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ \emptyset\}\ \ell_t \leftarrow^{sc} v_t \preceq_\pi \ell_s \leftarrow^{sc} v_s\ \left\{v_t', v_s'.\ v_t' = () * v_s' = () * \mathsf{exploit}\ \pi\ \emptyset\right\}$$

SIM-CALL-REVISED
$$\{\mathcal{V}\ v_t\ v_s * \mathsf{exploit}\ \pi\ \emptyset\}\ \mathsf{call}\ \mathsf{f}\ v_t \preceq_\pi \mathsf{call}\ \mathsf{f}\ v_s\ \left\{v_t', v_s'.\ \mathcal{V}\ v_t'\ v_s' * \mathsf{exploit}\ \pi\ \emptyset\right\}$$

Fig. 9. Some revised rules.

Returning ownership and revised rules. At this point, one may wonder what forces us to ever give up ownership of exploited locations. Intuitively, it is only sound to retain ownership until the thread *synchronizes* with other threads—if there is proper synchronization, there is no data race, and thus no UB. It turns out that for SimuLang, we only need to be concerned with atomic stores here: atomic loads do not release any information to other threads, so they cannot be used by those threads to avoid data races. Hence, the rule for atomic stores SIM-STORE-SC in Figure 9 requires giving up ownership of all exploited locations.[13] This is enforced via the precondition exploit $\pi\ \emptyset$: to establish exploit $\pi\ \emptyset$, one has to stop exploiting any location by applying RELEASE-EXPLOIT, which in turn requires giving up ownership of the exploited locations. exploit_frac determines which fraction needs to be returned. The values stored at these locations must be related by $\mathcal{V}$ to reestablish the invariant that all escaped non-exploited locations contain related values (similar to LOC-ESCAPE).

The exploit $\pi\ \emptyset$ precondition of SIM-STORE-SC also explains why the postcondition in the example from §3.2 needs to contain exploit $\pi\ \emptyset$: since we do not know the following code, it could perform an atomic store so we need to provide it with exploit $\pi\ \emptyset$. For similar reasons, the revised rule SIM-CALL-REVISED for calling unknown functions also has that precondition (and postcondition).

3.3 Combining Data-Race Exploitation and Coinductive Reasoning

To finish up the expository part of this paper, we revisit the example from §1. This will demonstrate the benefit of the unifying logic provided by Simuliris: we can seamlessly combine the reasoning principles presented in this section with the coinduction principles for loops from §2.4 to verify the challenging optimization in Figure 10, hoisting read accesses out of a (potentially diverging) loop.

The source program initializes a counter i and a result accumulator r. Then, it enters a loop where each iteration increments r by the value stored at y_s and increments i by 1, until i reaches the value stored at x_s. The optimized target program replaces the repeated loads of x_s and y_s with a single load for each location, using n and m to store the loaded values.

With the rules in Figure 9 and WHILE-COIND it is straightforward to verify the optimization in Figure 10: first, we use EXPLOIT-LOAD twice to gain ownership of x_s, x_t, y_s, and y_t.[14] Here, we leverage that, for EXPLOIT-LOAD to apply, it is sufficient for the non-atomic load to be *reachable* (via the relation $\rightarrow_?^*$). Intuitively, proving $e_s \rightarrow_?^* e_s'$ requires constructing a (thread-local) execution from e_s to e_s' starting in an arbitrary state. For this proof, one can assume that all executions of e_s are safe (*i.e.*, do not lead to UB). In the current example, both the load from y_s and from x_s can be

[13] We have also verified a stronger rule that allows retaining ownership over atomic stores if there is another non-atomic access reachable after the atomic store. This allows to *e.g.*, reorder non-atomic accesses before atomic stores.

[14] We only present the case where $x_s \neq y_s$. The $x_s = y_s$ case works the same except that we only use EXPLOIT-LOAD once.

$$\{x_t \leftrightarrow_h x_s * y_t \leftrightarrow_h y_s * \mathsf{exploit}\ \pi\ \emptyset\}$$

$$\begin{array}{lcl}
\mathsf{let}\ (\mathsf{n}, \mathsf{m}) := (!x_t, !y_t)\ \mathsf{in} & & \\
\mathsf{let}\ (\mathsf{i}, \mathsf{r}) := (\mathsf{ref}(0), \mathsf{ref}(\mathsf{m}))\ \mathsf{in} & & \mathsf{let}\ (\mathsf{i}, \mathsf{r}) := (\mathsf{ref}(0), \mathsf{ref}(!y_s))\ \mathsf{in} \\
\mathsf{while}\ !\mathsf{i} \neq \mathsf{n}\ \mathsf{do} & \preceq_\pi & \mathsf{while}\ !\mathsf{i} \neq !x_s\ \mathsf{do} \\
\quad i \leftarrow !\mathsf{i} + 1; \mathsf{r} \leftarrow !\mathsf{r} + \mathsf{m} & & \quad i \leftarrow !\mathsf{i} + 1; \mathsf{r} \leftarrow !\mathsf{r} + !y_s \\
\mathsf{od};\ !\mathsf{r} & & \mathsf{od};\ !\mathsf{r}
\end{array}$$

$$\{v_t, v_s.\ v_t = v_s * \mathsf{exploit}\ \pi\ \emptyset\}$$

Fig. 10. Simulation for the optimization from §1.

reached from the start of the program since both the code before the loop and the loop condition are executed unconditionally. For reaching $!x_s$, one has to skip over $!y_s$. This is allowed since one can assume that there is no undefined behavior in the source. One cannot assume anything about the value returned by $!y_s$ during the reachability proof, but this does not matter since $!x_s$ is executed regardless of which value is returned.

After acquiring ownership of x_s, x_t, y_s, and y_t, storing values v_s^x, v_t^x, v_s^y, and v_t^y with $\mathcal{V}\ v_t^x\ v_s^x$ and $\mathcal{V}\ v_t^y\ v_s^y$, the rest of the verification is standard separation logic reasoning: we begin with the loads from x_t and y_t in the target program, then we allocate i and r on both sides—using the ownership of y_s to justify $!y_s$. After that, we apply WHILE-COIND with the loop invariant:

$$\exists z_i, v_t^r, v_s^r.\ i_t \mapsto^{\mathrm{tgt}} z_i * i_s \mapsto^{\mathrm{src}} z_i * r_t \mapsto^{\mathrm{tgt}} v_t^r * r_s \mapsto^{\mathrm{src}} v_s^r * x_t \overset{q_x}{\mapsto}{}^{\mathrm{tgt}} v_t^x * x_s \overset{q_x}{\mapsto}{}^{\mathrm{src}} v_s^x * y_t \overset{q_y}{\mapsto}{}^{\mathrm{tgt}} v_t^y *$$
$$y_s \overset{q_y}{\mapsto}{}^{\mathrm{src}} v_s^y * \mathcal{V}\ v_t^r\ v_s^r * \mathcal{V}\ v_t^x\ v_s^x * \mathcal{V}\ v_t^y\ v_s^y * \mathsf{exploit}\ \pi\ (y_s \mapsto \mathsf{Rd}(q_y), x_s \mapsto \mathsf{Rd}(q_x))$$

This invariant might seem complicated, but it just contains ownership of all source and target locations and ensures that they point to related values. With this invariant, it is straightforward to verify the rest of the function. The optimization is justified because the loop invariant ensures that x_s always points to v_s^x when the condition executes, which is related by $\mathcal{V}$ to the value v_t^x used in the target program—this makes sure that the comparison evaluates the same in both source and target. A similar argument applies for optimizing the non-atomic load in the loop.

4 CONTEXTUAL REFINEMENT

We have seen how to prove simulations between source and target expressions with pre- and postconditions. However, for program transformations we are interested in a different relation: *contextual refinement* $e_t \sqsubseteq_{\mathrm{ctx}} e_s$. Contextual refinement ensures that a compiler can replace the (possibly) open expression e_s with the expression e_t in *any* part of the program without introducing additional behaviors. For instance, for the simple data race example (from §3.2) we want:

$$\mathsf{x} \leftarrow 42; !^{sc}\mathsf{y};\ 42 \ \sqsubseteq_{\mathrm{ctx}}\ \mathsf{x} \leftarrow 42; !^{sc}\mathsf{y};\ !\mathsf{x} \qquad \text{(DATA-RACE-CTX)}$$

where x and y are free variables. In the following, we define contextual refinement for SimuLang (in §4.1) and then show how it can be proven using the simulation relation $\{P\}\ e_t \preceq e_s\ \{\Phi\}$ (in §4.2).

4.1 Fair Termination-Preserving Contextual Refinement

To define contextual refinement, we start by defining whole-program refinement. This is basically standard, except that we have to account for our particular notion of "whole programs".

As already shown in Figure 2, a whole program is given by a list of function declarations. In Figure 11, we show an excerpt of how program execution is defined: the current machine configuration is given by a list of expressions (the *thread pool*) and the global state. The thread-pool

$$T \in \mathit{TPool} \triangleq \mathit{List}(\mathit{Expr})$$
$$\sigma \in \mathit{State} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{MemCell}$$
$$\mathit{Beh} \ni b ::= V(v) \mid \omega \mid \top$$

$$(K[\mathsf{fork}\{e\}], \sigma) \xrightarrow{\rho} (K[()], \sigma, [e])$$

$$\frac{(T(\pi), \sigma) \xrightarrow{\rho} (e', \sigma', \bar{e}_f)}{(T, \sigma) \xrightarrow{\rho}_{\text{tp}} (T[\pi \mapsto e'] + \bar{e}_f, \sigma')}$$

Fig. 11. Excerpt of the operational semantics of SimuLang.

step relation $\rightarrow_{\text{tp}}$ picks an arbitrary thread and performs a per-thread reduction step $\rightarrow$. (Here, $+\!\!+$ is list concatenation, adding the new threads to the end of the threadpool.) As an example of a per-thread step, we show the reduction rule for fork: the last component of the per-thread step relation indicates a list of new threads that are created by this step. All step relations are indexed by the list of functions ρ, which is relevant for call expressions.

To execute a given program ρ, we assume a fixed function name main indicating the entry point of the program, so that the initial machine configuration of a whole-program execution is given by $\mathcal{I} \triangleq ([\mathsf{call}\ \mathsf{main}\ ()], \emptyset)$: a single main thread, and the empty heap.[15] The (always non-empty) set of possible *behaviors* of this program is then defined by

$$\begin{aligned}\mathcal{B}(\rho) \triangleq & \left\{V(v) \mid \mathcal{I} \textit{ has a finite } \xrightarrow{\rho}_{\text{tp}} \textit{ execution where the main thread returns } v\right\}\\ \cup & \left\{\ \omega \ \mid \mathcal{I} \textit{ has a fair, infinite } \xrightarrow{\rho}_{\text{tp}} \textit{ execution}\right\}\\ \cup & \left\{\ \top \ \mid \mathcal{I} \textit{ can via } \xrightarrow{\rho}_{\text{tp}} \textit{ reach a configuration which is stuck}\right\}\end{aligned}$$

The last case defines when a program has *undefined behavior* (denoted by $\top$): we follow the standard approach of using *stuck states* to model undefined behavior. A program has reached a stuck state if one of its threads is neither a value nor can it take another step of reduction. Various kinds of safety violations cause programs to get stuck in our setting (*e.g.*, accessing unallocated memory, data races). Importantly, it is impossible to be stuck in a "good way": we use a non-blocking concurrency model, which means that there are no synchronization operations that halt the execution of one thread (in a good way) until another thread makes progress.

Moving to the case of a diverging program execution (denoted by ω), we crucially only consider *fair* infinite executions using the traditional definition of weak fairness [Lehmann et al. 1981]: an infinite execution of a thread pool is fair if every thread which is eventually always enabled does always eventually take a step. We consider a thread to be enabled when it can take a step:

$$\mathsf{enabled}(\rho, \sigma, T, \pi) \triangleq \exists e, e', \sigma'.\, T(\pi) = e \land (e, \sigma) \xrightarrow{\rho} (e', \sigma')$$

Since Simuliris uses a non-blocking concurrency model, this notion of fairness is equivalent to saying: each thread will either terminate in a value or take infinitely many steps.[16]

Refinement. We define a notion of *refinement* on behaviors as follows:

$$V(v_t) \sqsubseteq_{\text{beh}} V(v_s) \textit{ if } O(v_t, v_s) \qquad \omega \sqsubseteq_{\text{beh}} \omega \qquad b \sqsubseteq_{\text{beh}} \top$$

In particular, if a program has undefined behavior in the source, then *any* target behavior is allowed. Here, O defines the possible observations on return values. For SimuLang, this is mostly defined as equality, except that we assume that no observations are possible on locations, so $O(\ell_t, \ell_s)$ holds

[15] In our formal development, we also support non-empty initial heaps.

[16] This equivalence only holds for executions that do not get stuck, but programs that do get stuck anyway have undefined behavior.

for any two locations (as is standard). We can lift this notion of refinement to programs:

$$\rho_t \sqsubseteq_{\text{prog}} \rho_s \triangleq \forall b_{\text{t}} \in \mathcal{B}(\rho_t).\, \exists b_{\text{s}} \in \mathcal{B}(\rho_s).\, b_{\text{t}} \sqsubseteq_{\text{beh}} b_{\text{s}}$$

And finally, we lift this to contextual refinement of open terms by quantifying over an arbitrary closing context:

$$\begin{aligned} e_t \sqsubseteq_{\text{ctx}} e_s \triangleq \forall \rho, f, x, C.\, & (\mathsf{FreeVars}(C[e_t]) \cup \mathsf{FreeVars}(C[e_s]) \subseteq \{x\}) \wedge \mathsf{wf}(C) \wedge \\ & (\forall (f'\ x' \triangleq e') \in \rho.\, \mathsf{FreeVars}(e') \subseteq \{x'\} \wedge \mathsf{wf}(e')) \Rightarrow \\ & (f\ x \triangleq C[e_t], \rho) \sqsubseteq_{\text{prog}} (f\ x \triangleq C[e_s], \rho) \end{aligned}$$

The "closing context" consists of two parts: a context C (*i.e.*, an expression with a hole at an arbitrary place) to turn the expressions into function bodies (with the argument variable x as the only free variable), and a program ρ to supply all the other functions of the program (which must also be appropriately closed). Furthermore, both context and program must not contain any location values, which is captured by the well-formedness predicate wf.

4.2 Logical Relation

Now that we have defined contextual refinement, the question is—how can we prove it? Here we follow the usual recipe of (language-specific) *logical relations*. We already have a powerful simulation relation that works on closed expressions. We can lift it to open expressions by quantifying over a *closing substitution* γ that replaces free variables by related values—here we reuse the value relation $\mathcal{V}$ that we already used for external function calls:

$$e_t \leq_{\log} e_s \triangleq \forall \gamma : \mathsf{FreeVars}(e_t) \cup \mathsf{FreeVars}(e_s) \to \mathsf{Val} \times \mathsf{Val}.$$
$$\{\mathsf{exploit}\ \pi\ \emptyset * \forall x, v_t, v_s.\, \gamma(x) = (v_t, v_s) \Rightarrow \mathcal{V}\ v_t\ v_s\}\ \gamma_{\text{tgt}}(e_t) \preceq \gamma_{\text{src}}(e_s)\ \left\{v_t', v_s'.\, \mathsf{exploit}\ \pi\ \emptyset * \mathcal{V}\ v_t'\ v_s'\right\}$$

Here we use γ_{src} and γ_{tgt} to refer to the source and target projections of γ (which assigns two values to each free variable), respectively. The final values that the terms reduce to must also be in the value relation. As in SIM-CALL-REVISED, we also ensure no data-race exploitation is ongoing at the beginning and end of this simulation.

As one would expect from a logical relation, $\leq_{\log}$ is compatible with all expression formers, in the sense that when all subexpressions are related, then so are the compound expressions. This gives rise to the *fundamental theorem* of our logical relation, and the fact that it is a precongruence with respect to language contexts:

THEOREM 4.1 (FUNDAMENTAL THEOREM). *Let e be a well-formed expression. Then $e \leq_{log} e$.*

THEOREM 4.2 (CONTEXTUAL CLOSURE). *Assume $e_t \leq_{log} e_s$ and* $\mathsf{wf}(C)$. *Then $C[e_t] \leq_{log} C[e_s]$.*

As usual, the fundamental theorem establishes that the logical relation is reflexive—for terms that do not contain literal location values ℓ.

The core property of the logical relation is that it can be used to establish contextual refinement:

STATEMENT 4.3 (ADEQUACY). *If $e_t \leq_{log} e_s$, then $e_t \sqsubseteq_{ctx} e_s$.*

We prove Statement 4.3 in the next section (see Theorem 5.2).

How does this help to establish our contextual refinement example (DATA-RACE-CTX)? First we can apply adequacy, which turns the goal into $\mathsf{x} \leftarrow 42; !^{sc}\mathsf{y};\ 42 \ \leq_{\log}\ \mathsf{x} \leftarrow 42; !^{sc}\mathsf{y};\ !\mathsf{x}$. Now we unfold the definition of $\leq_{\log}$: the closing substitution γ will assign a source and target value to each free variable, and we learn that those values are related. The free variables of our example are x and y, so we obtain almost exactly the statement we proved in §3.2. The only difference is that we

$$\begin{aligned}
&\mathsf{sim}\ e_t\ e_s\ \Phi = \forall \sigma_t, \sigma_s, \rho_t, \rho_s.\, \mathcal{S}(\rho_t, \sigma_t, \rho_s, \sigma_s) * \mathsf{safe}(\rho_s, e_s, \sigma_s) \mathrel{-\!\!*} \Rrightarrow \\
&\Big(\textit{Base case:}\quad \exists e_s', \sigma_s'.\, (e_s, \sigma_s) \xrightarrow{\rho_s}{}^{*} (e_s', \sigma_s') * \mathcal{S}(\rho_t, \sigma_t, \rho_s, \sigma_s') * \Phi\ e_t\ e_s'\Big) \lor \\
&\Big(\textit{Step case:}\quad \mathsf{reducible}(\rho_t, e_t, \sigma_t) * \forall e_t', \sigma_t', T_t.\, (e_t, \sigma_t) \xrightarrow{\rho_t} (e_t', \sigma_t', T_t) \mathrel{-\!\!*} \Rrightarrow \\
&\qquad \big(\textit{Source stutter:}\quad T_t = [\,] * \mathcal{S}(\rho_t, \sigma_t', \rho_s, \sigma_s) * \mathsf{sim}\ e_t'\ e_s\ \Phi\big) \lor \\
&\qquad \big(\textit{Source step:}\quad \exists e_s', e_s'', \sigma_s', \sigma_s'', T_s.\, (e_s, \sigma_s) \xrightarrow{\rho_s}{}^{*} (e_s', \sigma_s') * (e_s', \sigma_s') \xrightarrow{\rho_s} (e_s'', \sigma_s'', T_s) * \\
&\qquad\qquad \mathcal{S}(\rho_t, \sigma_t', \rho_s, \sigma_s'') * \mathsf{sim}\ e_t'\ e_s''\ \Phi * |T_t| = |T_s| * \mathop{\Huge *}_{(e_t''', e_s''') \in \mathsf{zip}(T_t, T_s)} \mathsf{sim}\ e_t'''\ e_s'''\ \mathcal{V}\big)\Big)
\end{aligned}$$

Fig. 12. Simplified simulation weakest precondition.

only learn that x_t and x_s (and y_t and y_s) are in the value relation, not that they are related escaped locations—but the proof is easily adjusted to this.

In other words, to prove correctness of optimizations that replace e_s by e_t anywhere in the program, it is sufficient to establish $e_t \leq_{\log} e_s$—which we can in turn do with the proof techniques we have introduced in the previous two sections.

5 MODEL AND ADEQUACY

Now that we have stated adequacy, the question of course is: how can we prove it? To discuss this proof, we first explain how our relational Hoare quadruples are defined (in §5.1), and then we give an idea of the key lemma that enables the adequacy proof of the logical relation (in §5.2). Most of what we discuss in this section is defined in a language-generic way, so while the definitions and proofs are involved, they have to be done only once and can be reused many times.

5.1 Simulation Relation

The simulation relation $\{P\}\ e_t \preceq e_s\ \{\Phi\}$ forms the heart of Simuliris. It is itself defined in separation logic; more specifically, it is defined in a variant of Iris [Jung et al. 2018b] that we dub Iris$^{\text{light}}$. Iris$^{\text{light}}$ is essentially "Iris without the step-indexing": we keep the basic logic of bunched implications [O'Hearn and Pym 1999] and the persistence modality, but we ignore all the step-indexing aspects of Iris (*e.g.*, the later modality $\triangleright P$, guarded recursion, and impredicative invariants). Technically, instead of defining a new logic, we found it easier to literally reuse the model of Iris itself, but fix the step-index to be 0. This lets us directly use all of the Iris infrastructure.

Inspired by Iris's weakest precondition, we define the simulation relation as

$$\{P\}\ e_t \preceq e_s\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{sim}\ e_t\ e_s\ \Phi)$$

where $\mathsf{sim}\ e_t\ e_s\ \Phi$ is the weakest precondition that we need to impose such that e_s simulates e_t with postcondition Φ, and the persistence modality $\Box$ ensures that the simulation relation is duplicable (*i.e.*, the fact that e_s simulates e_t can be used arbitrarily many times). The interesting part of this definition is, of course, $\mathsf{sim}\ e_t\ e_s\ \Phi$ itself. A simplified[17] version of $\mathsf{sim}\ e_t\ e_s\ \Phi$ is shown in Figure 12, with a focus on our four essential features: *ownership reasoning*, *undefined behavior*, *concurrency*, and *fair termination preservation*.

[17] The simplified version glosses over two details: an extension for exploiting UB of data races (see §6) and an additional case for skipping function calls to obtain an open simulation. A complete definition can be found in the supplementary material [Gäher et al. 2022, §1].

We assume that the setup of the language roughly matches SimuLang in the sense that there is a per-thread step relation $\xrightarrow{\rho}$ indexed by some "program" that maps function names to "function bodies" (the definition of which is left up to the language). We write $(e, \sigma) \xrightarrow{\rho} (e', \sigma')$ for a step where no threads are forked off. Finally, we say that an expression e is *safe* in state σ and program ρ, written $\mathsf{safe}(\rho, e, \sigma)$, if it cannot reach a *stuck* configuration, *i.e.*, a state and expression which is neither a value nor reducible, in any number of steps.

We start the explanation of sim with a discussion of the general structure, before we focus on the four essential features. In general, the simulation weakest precondition factors into two cases: the *base case* and the *step case*. In the base case, we have reached the postcondition: we allow executing some more steps of the source expression (for target stuttering) and afterwards we prove the postcondition $\Phi\ e_t\ e_s'$, which is a separation-logic assertion and thus can also make statements about the final states. We do not generally require termination in the base case in order to enable WHILE-PACO, but termination can be stated in the postcondition by requiring e_t and e_s' to be values. In the step case, we have to prove that e_t is reducible and that for each target step we can find a matching source step—or perform source stuttering. In the source stutter case, we can relate the new target expression e_t' to *the same* source expression. In the source step case, we have to execute at least one source step (but we can execute more, which again would be target stuttering). The source has to fork exactly as many threads as the target step. We then have to show that the resulting target expression simulates the source expression.

Ownership reasoning. The support for ownership reasoning in Simuliris is inherited from Iris$^{\text{light}}$. In particular, the update modality $\Rrightarrow P$ expresses that we are allowed to perform *frame-preserving updates* on the ghost state in order to obtain the resources satisfying P. This incorporates the Iris approach to ghost state [Jung et al. 2018b] into Simuliris.

The state interpretation $\mathcal{S}(\rho_t, \sigma_t, \rho_s, \sigma_s)$ connects that ghost state to the physical states σ_t and σ_s of the target and source program. (This follows the same setup as typical Hoare triples in Iris.) For example, in SimuLang, the state interpretation governs which locations are currently local and which are escaped. It also ensures that all escaped locations store values related in the value relation $\mathcal{V}$. Additionally, it connects the target and source programs ρ_t and ρ_s to separation logic assertions that provide knowledge of a function's source code, to enable local reasoning when calling them. (This is not needed for the examples so we have omitted it from the paper.)

Concurrency. Our simulation relation enforces a one-to-one mapping between threads in the source and target programs: for each thread forked in the target, there must be a matching thread simulating it in the source. This is restrictive, but sufficient for the vast majority of compiler optimizations—and it is crucial to our proof of fair termination preservation.

Note how in the source step case, we use a separating conjunction to concisely express that the simulation proofs of the forked-off threads, and of the original thread, are all based on disjoint resources.[18] This ensures that the simulation of one thread does not interfere with the resources required by another thread.

Undefined behavior. We model UB via stuck expressions. Thus, the assumption that the source does not have UB is reflected in sim by assuming $\mathsf{safe}(\rho_s, e_s, \sigma_s)$. This is exploited by the $e_s \rightsquigarrow Q$ judgement that we have seen in §2.3, which is defined as:

$$e_s \rightsquigarrow Q \triangleq (\forall \rho_s, \sigma_s.\ \mathsf{safe}(\rho_s, e_s, \sigma_s) \Rightarrow Q)$$

[18]For the postcondition of the forked-off threads, we require the value relation $\mathcal{V}$, overloading the notation to implicitly lift it to a relation on expressions: $\mathcal{V}\ e_t\ e_s \triangleq \exists v_t, v_s.\ e_t = v_t * e_s = v_s * \mathcal{V}\ v_t\ v_s$.

Logic Level **Meta Level**

$$\begin{tikzcd}
e_t \leq_{\text{log}} e_s \arrow[r, dashed, "\text{Theorem 5.2}"] \arrow[d] & e_t \sqsubseteq_{\text{ctx}} e_s & \text{(language-specific)} \\
\rho_t \leq_{\text{prog}} \rho_s \arrow[r, "\text{Lemma 5.1}"'] & \rho_t \sqsubseteq_{\text{prog}} \rho_s \arrow[u] & \text{(language-independent)}
\end{tikzcd}$$

Fig. 13. Overview of the proof of Statement 4.3 (proven in Theorem 5.2) in terms of Lemma 5.1.

Fair termination preservation. Where do fairness and termination preservation factor into the definition? As already mentioned, we simulate threads pairwise. Thus, if we did a lockstep simulation, then fair termination preservation would be easy: for every target step, there would be exactly one source step of the corresponding thread. Hence, if the target diverges, then the source diverges—and both executions execute threads in the same order, so fairness is preserved.

What makes fairness and termination preservation challenging is *implicit stuttering*. We have seen that implicit stuttering is quite useful for reasoning about target and source individually without counting the steps (*e.g.*, see SOURCE-FOCUS). Here we have to pay the price: given a fair diverging execution of the target, we have to ensure that no source thread gets "forgotten" for too long such that the corresponding source execution is still fair.

To achieve this, we use a *mixed fixed-point* to define sim. More specifically, we nest two fixed-points: a greatest fixed-point and a least fixed-point. We define $\mathsf{sim} \triangleq \nu G. \mu L. \, \mathsf{simbody}(G, L)$ where simbody is what we obtain in Figure 12 if we replace the red occurrences of sim with G ("greatest fixed-point") and the blue occurrence with L ("least fixed-point"). The resulting definition satisfies $\mathsf{sim} = \mathsf{simbody}(\mathsf{sim}, \mathsf{sim})$, but it is neither the greatest nor the least fixed-point—it is in between.

The greatest fixed-point part of it gives rise to a language-independent parametric coinduction principle, from which language-specific rules such as WHILE-PACO can be derived. The least fixed-point on the inside ensures that we can stutter the source, but we can only do so finitely often, which will be crucial for the adequacy proof of fair termination preservation.

5.2 Adequacy

Equipped with the simulation relation, we turn to our main result, Statement 4.3, the adequacy of the logical relation. The proof proceeds in multiple steps: As a stepping stone, we define a logical *whole-program* relation $\rho_t \leq_{\text{prog}} \rho_s$ and show in Lemma 5.1 that it implies whole-program refinement. This part is the heart of our adequacy proof and is language-independent. Using the stepping stone, we show that $\leq_{\text{log}}$ implies $\sqsubseteq_{\text{ctx}}$ in Theorem 5.2. This part is fairly straight-forward and language-specific. The proof steps are sketched in Figure 13.

We define $\rho_t \leq_{\text{prog}} \rho_s$ by lifting the simulation relation to whole programs:

$$\rho_t \leq_{\text{prog}} \rho_s \triangleq \forall (f\ x \triangleq e_s) \in \rho_s. \, \exists (f\ x \triangleq e_t) \in \rho_t. \, \forall v_t, v_s. \, \{\mathcal{V}\ v_t\ v_s\}\ e_t[v_t/x] \preceq e_s[v_s/x]\ \{\mathcal{V}\}$$

That is, for every source function $f\ x \triangleq e_s$, there must be an implementation $f\ x \triangleq e_t$ in the target such that if we insert related values into the function bodies e_t and e_s, then the resulting target expression simulates the resulting source expression. (The definition of how to substitute the argument into the function body is left to the language.) This definition satisfies:

LEMMA 5.1 (WHOLE-PROGRAM ADEQUACY). *If $\mathcal{S}(\rho_t, \emptyset, \rho_s, \emptyset) * \rho_t \leq_{prog} \rho_s$, then $\rho_t \sqsubseteq_{prog} \rho_s$.*

PROOF SKETCH. The proof of this lemma factors into two steps. Step one is concerned with the fact that sim is an *open simulation* that can skip matching function calls (SIM-CALL), which we omitted from Figure 12. We prove that this open simulation implies a *closed simulation* that looks almost exactly like Figure 12. This proof ties the big recursive knot over all the mutually recursive function definitions in the source and target programs. It essentially "inlines" the simulation of the calls that were skipped. Since open simulations are a well-known technique [Kang et al. 2015; Hur et al. 2012], we do not focus on them here.

In the second step, we prove that the closed simulation implies whole-program refinement. Roughly speaking, this step factors into three cases (corresponding to the cases of $\mathcal{B}(\rho)$):

(1) *Undefined Behavior.* To prove that the target does not have UB, we leverage reducibility in the step case in Figure 12. That is, the target will never be stuck, because all the expressions it can reach are reducible.
(2) *Result Value.* To prove that the target only terminates with values that are also possible in the source, we leverage the postcondition: if the target terminates, then the postcondition ensures that the source must also terminate in a value and both satisfy $\mathcal{V}$. The value relation, in turn, implies that the result values are related by O.
(3) *Fair Divergence.* Proving that a fair diverging target execution implies a fair diverging source execution is by far the most challenging case. Intuitively, we are termination-preserving because we do not allow infinite stuttering (since stuttering uses a *least* fixpoint), and we are *fair* termination-preserving because we simulate threads in a one-to-one correspondence. Unfortunately, there is a caveat to this intuitive argument—in a sense, through stuttering, the order in which threads are executed can change. For example, we can delay the execution of some source steps in one thread until steps in another thread have passed. Nevertheless, the heart of the intuitive argument can still be recovered (with a coinduction and three nested inductions)[19] and, hence, we obtain fair termination preservation. □

We now have all the pieces that we need to prove our main theorem, following the structure laid out in Figure 13 (reading the arrows backwards, since we are doing a goal-directed proof).

Theorem 5.2 (Adequacy, Statement 4.3). *If $e_t \leq_{log} e_s$, then $e_t \sqsubseteq_{ctx} e_s$.*

Proof Sketch. We assume some well-formed closing program ρ and a closing context C. Let us define $\rho_t \triangleq (f\ x \triangleq C[e_t]), \rho$ and $\rho_s \triangleq (f\ x \triangleq C[e_s]), \rho$. It remains to show $\rho_t \sqsubseteq_{\text{prog}} \rho_s$. With Lemma 5.1, it suffices to show $\mathcal{S}(\rho_t, \emptyset, \rho_s, \emptyset) * \rho_t \leq_{\text{prog}} \rho_s$. A proof of the initial state interpretation $\mathcal{S}(\rho_t, \emptyset, \rho_s, \emptyset)$ is assumed to be supplied by the specific language instance of Simuliris. The required whole-program refinement follows from $e_t \leq_{\text{log}} e_s$ and contextual closure of $\leq_{\text{log}}$ (for the function f), and from reflexivity of the logical relation (for all the other functions). □

6 SOUNDNESS OF EXPLOITING NON-ATOMIC ACCESSES

This section explains the core of the soundness argument for the rules that exploit undefined behavior of non-atomic accesses in SimuLang for optimizations from §3.

The state interpretation. To understand how exploit-store hands out ownership of escaped locations, we first need to see how the ownership of escaped locations is managed in general. The basic idea is that we maintain a *bijection* between the escaped locations in the source and target programs, and ensure that matching locations carry related values. This is expressed by the $\mathsf{heapbij}(H)$ predicate that is part of the state interpretation $\mathcal{S}$ and is (roughly) defined as follows:[20]

$$\mathsf{heapbij}(H) \triangleq \exists L.\ \mathsf{bijauth}(L) * \mathop{\Large *}_{(\ell_t,\ell_s)\in L} \exists q_h, v_t, v_s.\ H(\ell_s, q_h) * \mathcal{V}\ v_t\ v_s * \ell_t \overset{q_h}{\mapsto}{}^{\text{tgt}}\ v_t * \ell_s \overset{q_h}{\mapsto}{}^{\text{src}}\ v_s$$

Here, the set L is the bijection containing the escaped locations. The assertion $\mathsf{bijauth}(L)$ represents ownership of Iris ghost state which tracks full knowledge of the current bijection. The $\ell_t \leftrightarrow_h \ell_s$ that

[19]For space reasons, we are omitting the details from the paper.

[20]Technically, heapbij uses a slightly non-standard notion of points-to predicates that is defined for $q = 0$ (as True).

we have already seen talks about the same ghost state and reflects knowledge that (ℓ_t, ℓ_s) is in the bijection L. This is realized with the standard Iris mechanism of "authoritative ghost state" [Jung et al. 2018b]; the details of this technique do not matter here.

heapbij further says that for every pair of locations in L, we have ownership of those locations and their values are in the value relation. Since heapbij is part of the state interpretation $\mathcal{S}$, each thread can use this ownership to justify executing a step, but it has to give it back at the end of the instruction and cannot keep the ownership between instructions.

However, to justify EXPLOIT-STORE and EXPLOIT-LOAD, we need to remove (some part of) this ownership from heapbij. This is the purpose of the H parameter of heapbij: this relation determines which fraction q_h is stored in heapbij for each escaped location ℓ_s. A simple state interpretation would just use $H(\ell_s, q_h) \triangleq q_h = 1$ and for such a state interpretation one can prove SIM-LOAD-ESCAPED from §2.2. However, for exploiting data races we need a more complicated H that lets us store less than the full fraction for exploited locations. In particular, assume that the list $\overline{C}$ contains the collections of exploited locations for all threads (*i.e.*, the collection of thread π is $\overline{C}_\pi$ and is exposed through the assertion exploit $\pi\ \overline{C}_\pi$). Then, for each location ℓ_s, the fraction q_h stored in heapbij and the fractions handed out through exploitation should sum up to 1:[21]

$$H_{\overline{C}}(\ell_s, q_h) \triangleq 1 = q_h + \sum_\pi \begin{cases} \mathsf{exploit\_frac}(c) & \text{if } \overline{C}_\pi(\ell_s) = c \\ 0 & \text{otherwise} \end{cases}$$

While the above choice of H allows us to remove ownership from heapbij, this does not come for free. In particular, the ownership inside heapbij was used to justify rules like SIM-LOAD-ESCAPED. How can such a rule still be sound if there is no ownership in heapbij? This is where data races come in. The basic idea is simple: we can only remove ownership from heapbij if we prove that each access expecting this ownership to be in heapbij has a data race and thus need not be considered.

Concretely, consider the case where thread π uses EXPLOIT-STORE to remove the ownership of a location ℓ_s from heapbij. Now another thread π' tries to load ℓ_s but does not find the necessary ownership to justify the load in heapbij. So instead we show that there must be an execution with a data race: we know that thread π can reach a non-atomic store (due to the premise of EXPLOIT-STORE) and thread π' is ready to perform a load—and these two accesses form a data race, so we are done!

To make this argument formal, we need to make sure that for each c such that $\overline{C}_\pi(\ell_s) = c$ there is a safe configuration (*i.e.*, a thread pool T and a state σ), where thread π can reach a non-atomic store (if $c = \mathsf{Wr}$) or a non-atomic load (if $c = \mathsf{Rd}(q)$).[22] That safe configuration must be the current source configuration, except that thread π can differ. Formally speaking:

$$\begin{aligned}
&\mathsf{exploit_wf}(\rho_s, \sigma_s, T_s, \overline{C}) \triangleq \forall \pi, \ell_s, c.\ \overline{C}_\pi(\ell_s) = c \Rightarrow \\
&\quad \exists K, e, v.\ \mathsf{pool_safe}(\rho_s, T_s[\pi \mapsto e], \sigma_s) \land e \rightarrow_?^* K[c = \mathsf{Wr}\ ?\ \ell_s \leftarrow v\ :\ !\ell_s] \\
&e_s \rightarrow_?^* e_s' \triangleq \forall \rho_s, \sigma_s.\ \mathsf{safe}(\rho_s, e_s, \sigma_s) \Rightarrow \exists \sigma_s'.\ (e_s, \sigma_s) \xrightarrow{\rho_s}{}^* (e_s', \sigma_s')
\end{aligned}$$

One can view exploit_wf as keeping track of *alternative interleavings* where thread π is "paused" while the other source threads step along with the target. Since all interleavings must be data-race free, demonstrating a race in one of these alternative interleavings is sufficient. The reachability relation $\rightarrow_?^*$ is defined as described in §3.3, *i.e.*, e_s' is reachable from e_s via a thread-local execution from an arbitrary state under the assumption that e_s is safe.

In the above scenario where the thread π has exploited ℓ_s with $\overline{C}_\pi(\ell_s) = \mathsf{Wr}$ and another thread π' is ready to perform a load of ℓ_s (*i.e.*, $T_s(\pi') = K'[!\ell_s]$), exploit_wf implies $\mathsf{pool\_safe}(\rho_s, T_s[\pi' \mapsto$

[21] The additional $\overline{C}$ parameter of this H is fixed before it is passed to heapbij as shown later.
[22] $\mathsf{pool\_safe}(\rho_s, T_s, \sigma_s)$ is similar to $\mathsf{safe}(\rho_s, e_s, \sigma_s)$ except for a thread pool T_s instead of an expression e_s.

$K'[!\ell_s]][\pi \mapsto K[\ell_s \leftarrow v]], \sigma_s')$ (by executing thread π to reach the store). This supposedly safe thread pool has a data race between the load and the store, and thus is not safe—a contradiction!

To finish the proof of the rule for loading from escaped locations, we need to consider one more case: what if the thread π that exploited ℓ_s is also the thread π' performing the load, *i.e.*, $\pi = \pi'$? Then there is no data race as a thread cannot race with itself. Thus, the load rule cannot apply in this situation. We hence weaken SIM-LOAD-ESCAPED to not permit loading from locations that are being exploited by the current thread and arrive at SIM-LOAD-NA. This argument also works for atomic loads and thus also gives us a proof of SIM-LOAD-SC.

Now we have all the ingredients to define the state interpretation $\mathcal{S}$:

$$\mathcal{S}(\ldots, \rho_s, \sigma_s, T_s) \triangleq \ldots * \exists \overline{C}.\, \mathsf{exploitauth}(\overline{C}) * \mathsf{exploit\_wf}(\rho_s, \sigma_s, T_s, \overline{C}) * \mathsf{heapbij}(H_{\overline{C}})$$

We make use of the fact that our full simulation relation gives the state interpretation access to the source program ρ_s and the current source thread pool T_s. We omit the parts of the state interpretation that are responsible for defining the points-to predicates. C is linked to the exploit π C assertions via the authoritative $\mathsf{exploitauth}(\overline{C})$. exploit_wf and heapbij appear as described above.

Proving EXPLOIT-STORE. With the state interpretation in hand, let us see how we can prove EXPLOIT-STORE (EXPLOIT-LOAD is analogous), assuming exploit π C and $\ell_t \leftrightarrow_h \ell_s$ from the precondition as well as $C(\ell_s) = \bot$ and $e_s \rightarrow_{?}^{*} K[\, \ell_s \leftarrow v_0 \,]$. There are two cases to consider.

Suppose no thread has exploited ℓ_s. In this case we know that heapbij contains the full ownership of ℓ_s and ℓ_t. We add $\ell_s \mapsto \mathsf{Wr}$ to C which allows us to remove this ownership from heapbij and we can establish exploit_wf for the modified collection by adding $\mathsf{pool\_safe}(\rho_s, T_s, \sigma_s)$ for the current thread pool and source state.[23]

Alternatively, there is another thread that has exploited ℓ_s (we know that this cannot be the current thread thanks to the $C(\ell_s) = \bot$ precondition). But then we can construct a data race between the non-atomic access used to justify the exploitation by the other thread and the non-atomic access given to EXPLOIT-STORE and we are done.

Maintaining the state interpretation. We have seen proof sketches for some of the rules, but there is an important proof obligation that we have not discussed so far: for each operation we need to prove that it maintains the state interpretation. Since the state interpretation is parametrized by the thread pool, this is non-trivial even for pure operations! In particular, for reestablishing $\mathsf{exploit\_wf}(\rho_s, \sigma_s', T_s', \overline{C})$ for the new thread pool T_s' and state σ_s', we need to update the $\mathsf{pool\_safe}(\rho_s, T_s'[\pi \mapsto e], \sigma_s')$ assertions for exploited locations (*i.e.*, making sure that we can still construct data races for conflicting accesses).

For a pure step in thread π', we consider two cases: if $\pi \neq \pi'$, we can just replay the step in the alternative configuration of $\mathsf{pool\_safe}(\rho_s, T_s[\pi \mapsto e], \sigma_s)$. (Basically, we are reordering this step around the exploited non-atomic access.) If $\pi = \pi'$, we can ignore this step since only the *other* threads matter ($T_s[\pi \mapsto e]$ overwrites the current thread) and the state σ_s do not change.

However, the latter case is more complicated for steps that interact with the heap and where possibly $\sigma_s' \neq \sigma_s$. Most of these operations can be dealt with via careful extensions of exploit_wf (see Gäher et al. [2022, §4]). Here we will focus on the most interesting case, which is an atomic store, *i.e.*, maintaining the invariant when proving SIM-STORE-SC. This will also show why this rule is only provable for exploit π' $\emptyset$ (assuming the store is executed in thread π'). First, we can assume that no other thread is exploiting the location ℓ_s as there would be a data race otherwise. This means that heapbij contains full ownership of ℓ_t and ℓ_s, which we use to justify the store in source and target. It is also easy to reestablish $\mathsf{pool\_safe}(\rho_s, T_s'[\pi \mapsto e], \sigma_s')$ for threads $\pi \neq \pi'$ by

[23] The full simulation relation uses $\mathsf{pool\_safe}(\rho_s, T_s, \sigma_s)$ instead of $\mathsf{safe}(\rho_s, e_s, \sigma_s)$ as shown in §5.

```
*x = 42;                // x: &mut i32
let r = *x;        ≤    *x = 42;
f();                    f();
return r;               return *x;
```

(a) Moving a load across unknown code.

```
let r = *x;             // x: &i32
while f(r) {       ≤    while f(*x) {
  g();                    g();
}                       }
```

(b) A new loop hoisting optimization.

Fig. 14. Two examples of Stacked Borrows optimizations done in Simuliris.

replaying the store. However, there is a problem when updating the pool_safe($\rho_s, T_s[\pi' \mapsto e], \sigma_s$) to pool_safe($\rho_s, T_s'[\pi' \mapsto e], \sigma_s'$) for exploited locations of the current thread ($\pi = \pi'$): the current expression of the thread π' in the alternative execution is e and not the atomic store, so we cannot just mirror the atomic store in the alternative execution. But without this we have a problem as the source state is not in sync any more! The only way out is to require that the current thread does not exploit any locations, which leads to the exploit π' $\emptyset$ precondition of SIM-STORE-SC. For non-atomic stores, we similarly cannot update the state in the pool_safe for locations exploited by the current thread. However, this can be fixed by tweaking the definition of exploit_wf as any thread that would observe this difference in states would race with the non-atomic store.

7 SIMULIRIS MEETS STACKED BORROWS

To demonstrate the flexibility of Simuliris as a language-generic framework, we instantiate it with the language of Stacked Borrows [Jung et al. 2020], a memory model for Rust proposed to enable aliasing-based optimizations. In §7.1, we give a brief overview of Stacked Borrows to explain how the correctness proofs of such optimizations benefit from Simuliris. In particular, Simuliris helped a lot when extending the soundness proofs of these optimizations to *concurrency* (with sequentially consistent accesses). We verify correctness not only of the original paper's optimizations, but also of a new loop hoisting optimization that makes use of Simuliris's support for coinduction.

7.1 Stacked Borrows: An Aliasing Model for Rust

Rust is a systems programming language that gives strong static guarantees through an ownership-based type system and enforces an aliasing principle of *Aliasing XOR Mutability* (AXM): at any point in time, data in memory either has one *mutable reference* `&mut T` that is *unique* and allows mutation, or multiple *shared references* `&T` that only allow read access.

In principle, these guarantees can enable aliasing-based optimizations, such as the one given in Figure 14a ("Example 1" in [Jung et al. 2020]). It works on the body of a Rust function whose argument is a mutable reference `x: &mut i32` to a 32-bit signed integer, and whose return type is `i32`. As the function takes a mutable reference `x`, by AXM, the unknown function `f` should not have an alias to the memory location referenced by `x`. Thus, the call to `f()` should not change the value of x, making it safe to move the load from `x` up across the call. (And this in turn should enable constant propagation of `42` to avoid the load altogether.)

But there is a problem: Rust makes heavy use of `unsafe` code. The purpose of `unsafe` code is to let programmers write code that is correct for reasons that are too subtle for the compiler to understand. However, this also means that Rust's type system guarantees are not implicitly upheld any more when `unsafe` code is being used. We thus need to explicitly demand that `unsafe` code follows the aliasing discipline. Stacked Borrows makes this precise, while still allowing pointer-based data structures with heavy aliasing to be implemented in `unsafe` code. The basic idea is to associate each reference with a *tag*, and each location with a *borrow stack* of tags. The stack tracks which tags may access this location; using a reference with a different tag is undefined behavior.

The original Stacked Borrows formalization verifies several optimizations in a *sequential* language with an ad-hoc ownership-based coinductive open simulation relation not unlike the model of Simuliris. More specifically, the correctness of an optimization relies on the notion of ownership of some tag t. In the example in Figure 14a, when the function gets the argument `x` as a mutable reference to a location ℓ_x, it also acquires the ownership $\mathsf{OwnTag}(t_x)$ of a unique tag t_x associated with `x`. $\mathsf{OwnTag}(t_x)$ intuitively enforces that the tag t_x is at the top of ℓ_x's stack. The correctness argument then relies on two points: (i) as the function body keeps $\mathsf{OwnTag}(t_x)$ throughout its operations, the function `f` has no ownership of t_x, and so `f` cannot use the tag t_x to access ℓ_x; and (ii) `f` cannot use any other tag to access ℓ_x either, as such an access would pop t_x from ℓ_x's stack, making the load `*x` in the source (right after the call of `f`) undefined behavior. Ultimately the ownership of $\mathsf{OwnTag}(t_x)$ prevents `f` from accessing ℓ_x, so moving the load up is sound.

7.2 Stacked Borrows in Simuliris

The use of ownership-based reasoning makes Stacked Borrows an excellent candidate application of Simuliris. We have instantiated Simuliris with the Stacked Borrows language, derived a logic for proving optimizations (closely following the original setup), and ported all the optimization proofs. The place where Simuliris shines is that a lot of the simulation infrastructure can be shared with SimuLang (whole-program adequacy, coinduction, basic structural rules), and that we obtain a proper separation logic with an interactive proof mode for carrying out proofs of optimizations without directly reasoning about the underlying model of resources.

Using this infrastructure, it was easy to prove a new loop hoisting optimization (Figure 14b). Here, `f` and `g` are closures and `x` is an immutable shared reference. We have proven that the repeated loads `*x` in each iteration can be replaced by a single load before the loop. The intuition is that ownership of the tag t_x for the shared reference `x` can be maintained through the entire loop.

A concurrent version of Stacked Borrows. Jung et al. [2020] only consider a sequential language. We have extended the Stacked Borrows language with support for concurrency and shown that the original optimizations are still correct. As part of this we discovered that a direct port of the original semantics to a concurrent setting would not give the desired results: with the original Stacked Borrows semantics, load instructions (*e.g.*, `*x`) directly trigger UB when the tag is not in the borrow stack. In a concurrent setting, this choice invalidates moving loads up like in Figure 14a: if the tag is removed from the stack by a concurrent thread, the target would trigger UB at the load in line 2 before `f` is even called, whereas the source would avoid UB entirely if `f` never returns (and the load `*x` is never performed). Thus the optimization may introduce new UB!

We have thus relaxed the original semantics: instead of directly triggering UB when doing a load with an invalid tag, the load returns *poison*, a special value which only triggers UB when it is being used (a form of *deferred UB* [Lee et al. 2017]). We can elegantly reflect this into separation logic: when the target load from ℓ yields poison, we obtain ownership of the assertion $\mathsf{Tainted}(t, \ell)$ stating that a tag t can never be contained in the source stack for ℓ again. Later in the proof, we use this assertion to prove that the source also loads poison, so both executions are in sync again.

8 RELATED WORK

Simuliris is, to our knowledge, the first separation logic designed for verifying concurrent compiler optimizations. This brings together two broad lines of research, of which we discuss the most closely related work here. Also see Figure 1 in the introduction for an overview.

Concurrent compiler optimizations. As already mentioned in the introduction, there are several projects with the goal of equipping CompCert with support for concurrency. However, even for purely sequential code, CompCert will not hoist loads from escaped pointers out of a loop (which is quite hard to do in a general optimization algorithm, a problem that we are side-stepping by considering only concrete optimizations). Correspondingly, our main motivating example (§1) is not (to our knowledge) verified by any of these variants of CompCert.

That said, the infrastructure for reasoning about sequential code in CompCert is in some ways more powerful than what we built for SimuLang. For example, CompCert's "memory injection", which corresponds to our heap bijection managing escaped locations, supports mapping multiple memory blocks of the source program into a single target block, whereas we enforce a one-to-one mapping of source and target blocks. On the other hand, since our bijection is managed via separation logic ghost state, it is arguably more convenient to work with. We leave it to future work to find a nice separation logic interface for the full power of CompCert's memory injection.

Most concurrent variants of CompCert establish a termination-preserving refinement, omitting the fairness constraint on infinite executions. While we are not aware of examples where CompCert would violate fair termination preservation, this property is not formally established. Likewise, we do not establish that our simulation relation is termination-preserving for the (unrealistic) case of unfair infinite executions, but we believe it is. (These two notions of correctness are incomparable.)

CASCompCert [Jiang et al. 2019] and Concurrent CompCert [Beringer et al. 2014; Cuellar 2020] show that sequential reasoning can be applied to non-synchronizing fragments of concurrent programs. This should, in principle, be sufficient to verify our main example using sequential reasoning only (except, as already mentioned, CompCert does not perform this optimization even for sequential code). However, in both of these approaches, the sequential semantics that they use for reasoning about program transformations does not model synchronization directly—rather, synchronization is performed as a side-effect of opaque external function calls. This rules out any optimization for which one has to reason in detail about the behavior of a synchronizing operation, such as in our example in §3.2.

The Concurrent Abstraction Layers [Gu et al. 2018] variant of CompCertX [Gu et al. 2015] is (to our knowledge) the only concurrent variant of CompCert that establishes *fair* termination preservation. Their "push/pull" model of shared memory is akin to our data-race detector, though we do not need a "global log". However, in their work, *all* interactions with shared memory (not just synchronization, *i.e.*, atomic accesses) are handled via external function calls. This appears to rule out even the most basic optimizations on non-atomic accesses.

CompCertTSO [Ševčík et al. 2013] uses the same approach as ours: they build concurrency into the operational semantics. However, the TSO (total store order) model of concurrency gives well-defined behavior to racy programs, and thus does not permit optimizations such as the example from the introduction.

Besides the work on CompCert, there also has been a lot of work on models of weak-memory concurrency, and the set of supported program transformations is a relevant point of comparison in this space. In particular, some of that work verifies correctness of various memory access reorderings [Ševčík 2011; Morisset et al. 2013; Vafeiadis et al. 2015; Kang et al. 2017]. Those models often consider a larger subset of C11 atomics (whereas we just support non-atomic and sequentially consistent accesses). However, all of that work only considers finite program executions, *i.e.*, they do not verify termination preservation. They also only verify reorderings of immediately adjacent instructions and do not formally prove that this is sufficient for more complex optimizations such as hoisting an instruction out of a loop. We have verified all reorderings established by this prior work that involve only the accesses supported by our system and at least one non-atomic access (*i.e.*, the reorderings that are related to data races); see our supplementary material [Gäher et al. 2022] for details. We have also verified one more reordering that has *not* been verified in prior work: moving a non-atomic load before a sequentially consistent load. This optimization is correct in our system, but it is interestingly incorrect in memory models with C11-style relaxed accesses—to account for the particularly weak behavior of ARM processors (which ignore control dependencies between loads), the definition of data races around relaxed accesses has the side-effect that this optimization can *introduce* a race into a previously race-free program, rendering it invalid.

Separation logic for contextual refinement. The idea of using separation logic reasoning in a simulation relation has been explored before. CompCert contains a separation-logic-style library for heap predicates[24] to simplify reasoning about the shape of stack frames—but this library does not provide a full-fledged logic. Other logics for simulations have mostly been developed in the context of using refinement for *program verification*: the goal is to show that an *implementation* of an abstract data type implements a *specification*. As such, this line of work lacks the ability to exploit undefined behavior, as we require for our example. However, that is not the only limitation.

The line of work by Liang et al. [2014]; Liang and Feng [2016, 2018] on rely-guarantee style relational separation logics supports concurrency and fair termination preservation. Unlike Simuliris, this work supports blocking operations [Liang and Feng 2018] and can be used to reason explicitly about fairness assumptions, such as when proving that an efficient, concurrent implementation with spinlocks implements a particular specification. However, it does not support passing pointers between the context and library (as would be the case in our motivating example). Moreover, this work has not been mechanized.

Another family of work uses Iris [Jung et al. 2015, 2016, 2018b; Krebbers et al. 2017a], a framework for defining concurrent separation logics with a flexible form of "ghost ownership", as the logical basis for refinement proofs. In particular, ReLoC [Frumin et al. 2018] shows how to define a binary logical relation in the (usually unary) Iris program logic, and establishes a number of contextual refinement results. However, Iris uses the technique of *step-indexing* to define *impredicative invariants* [Svendsen and Birkedal 2014], which are used by lots of prior work to great effect, but which come with a serious downside: step-indexing implies that Iris can only be used to reason about safety properties, not liveness properties such as termination preservation. In ReLoC, a diverging implementation refines all specification programs. Tassarotti et al. [2017] bend that limitation by showing that Iris can in fact be used to establish fair termination preservation under the assumption that the non-determinism in the language under consideration is finitely bounded.

Recent work by Spies et al. [2021] shows that *transfinite* step-indexing permits the use of Iris for verifying liveness properties such as termination without such an assumption, but this work has so far only been applied to sequential programs.

In our work, we avoid the need for step-indexing in the first place. The main feature enabled by step-indexing in Iris is a very modular treatment of invariants: any part of the proof can establish an arbitrary invariant on shared state and then communicate knowledge about that invariant to other threads or functions. In contrast, Simuliris only requires a single global invariant (namely, that escaped pointers point to related values in the source and target heaps). With all invariants being statically determined upfront, there is— for now—no need for step-indexing in Simuliris. In the future, it would be interesting to explore which new reasoning principles can be obtained in Simuliris by making use of transfinite step-indexing.

9 LIMITATIONS AND FUTURE WORK

Fairness reasoning. In Section 5.2, we prove that our logical relation $e_t \leq_{\log} e_s$ implies *fair* termination-preserving contextual refinement. For the fairness part of the proof, we get to assume that the target is executed on a fair scheduler. This assumption *is not* reflected into the logic and, hence, we also *have no* reasoning principles in the logic which exploit the fair scheduling of the target. Instead, we use the assumption in the adequacy proof to show that the simulated source

[24] See https://compcert.org/doc/html/compcert.common.Separation.html.

execution must also be fair. This way, we are guaranteed that our simulation relation cannot be used to verify bogus optimizations, *e.g.*, by exploiting a single unfair, diverging source execution to make a program always diverge (*e.g.*, replacing a spinlock by a trivially diverging program).

However, Simuliris cannot be used to show, for example, that it is sound to replace a sequential program with a more efficient, concurrent implementation which uses spinlocks to synchronize. The goal of Simuliris is to verify compiler transformations, and we are not aware of any compiler that would automatically perform such optimizations. Hence, we have opted for not exposing the fairness assumption into the logic, making the logic less powerful but also simpler to use by avoiding the bookkeeping typically needed to reason directly about fairness.

Whole-program optimizations. In Theorem 5.2, we show that our logical relation implies a contextual refinement, which is our top-level soundness result. Due to the nature of contextual refinement, which requires correctness of optimizations for any surrounding context, Simuliris can currently not be used for optimizations that require knowledge of the entire program (*e.g.*, removing global variables). However, since the core of this result, Lemma 5.1, states a property about whole programs, Simuliris could conceivably be extended to verify whole-program optimizations.

Optimization algorithms. In this work, we have verified concrete *concurrent program transformations*. We have not verified optimization algorithms (which automatically apply program transformations based on a program analysis), as would be required in a compiler correctness proof. Verifying an actual optimization pass is a bigger task than just verifying a few particular program transformations in isolation: one needs to prove a simulation not for a concrete piece of code, but for any possible result of the optimization algorithm. In the future, it would be interesting to explore the integration of Simuliris into a compiler correctness proof. This poses a number of challenges, for instance on the question of when to escape locations: for verifying individual optimizations, we can escape locations as soon as possible, but for general verification of optimization algorithms, it can be more sensible to delay escaping until it is necessary, thus escaping complicated structures at once [Stewart et al. 2015]. While our logic supports escaping tree-shaped data structures using induction, the current rules do not allow escaping cyclic structures like doubly-linked lists (but we think the existing model would permit us to add such reasoning principles).

Weak memory models. SimuLang provides a memory model with sequentially consistent and non-atomic accesses. It would be interesting to extend this work to weak memory models like the C11 model that provide more kinds of accesses (*e.g.*, relaxed accesses). One challenge here will be adapting the reasoning principles for exploiting data-race undefined behavior, since (as discussed in §8) the addition of relaxed accesses makes some currently supported optimizations unsound.

I/O. Our model languages do not have support for I/O. An interesting future direction would be to integrate Simuliris with recent advances in reasoning about I/O (in particular, interaction trees [Xia et al. 2020]) to prove correctness of optimizations in the presence of I/O.

ACKNOWLEDGMENTS

We thank Ori Lahav and Viktor Vafeiadis for explanations of data races in weak memory models, as well as our shepherd Hongjin Liang and the anonymous reviewers for their helpful feedback. This research was supported, in part, by European Research Council (ERC) Consolidator Grants for the projects "RustBelt" and "PERSIST", funded under the European Union's Horizon 2020 Framework Programme (grant agreements no. 683289 and 101003349, respectively), by a Google PhD Fellowship for the second author, by the Dutch Research Council (NWO), project 016.Veni.192.259, by the International Max Planck Research School on Trustworthy Computing (IMPRS-TRUST), and by generous awards from Android Security's ASPIRE program.

REFERENCES

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL*. 14–25. https://doi.org/10.1145/964001.964003

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP (LNCS, Vol. 8410)*. 107–127. https://doi.org/10.1007/978-3-642-54833-8_7

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*. 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*. 55–72.

Santiago Cuellar. 2020. *Concurrent Permission Machine for modular proofs of optimizing compilers with shared memory concurrency*. Ph.D. Dissertation. Princeton University, New Jersey, USA. https://dataspace.princeton.edu/handle/88435/dsp01qr46r378d

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. 646–661. https://doi.org/10.1145/3192366.3192381

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang(-)Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: Technical Documentation and Coq Development. https://doi.org/10.5281/zenodo.5667545

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. 59–72. https://doi.org/10.1145/2103656.2103666

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. In *POPL*. 193–206. https://doi.org/10.1145/2429069.2429093

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards certified separate compilation for concurrent programs. In *PLDI*. 111–125. https://doi.org/10.1145/3314221.3314595

Ralf Jung. 2020. *Understanding and evolving the Rust programming language*. Ph.D. Dissertation. Universität des Saarlandes, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. 175–189. https://doi.org/10.1145/3009837.3009850

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI*. 326–335. https://doi.org/10.1145/2737924.2738005

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. 179–192. https://doi.org/10.1145/2535838.2535841

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *PLDI*. 633–647. https://doi.org/10.1145/3062341.3062343

D. Lehmann, A. Pnueli, and J. Stavi. 1981. Impartiality, justice and fairness: The ethics of concurrent termination. In *Automata, Languages and Programming*. 264–277. https://doi.org/10.1007/3-540-10843-2_22

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. 42–54. https://doi.org/10.1145/1111037.1111042

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *POPL*. 385–399. https://doi.org/10.1145/2837614.2837635

Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. *PACMPL* 2, POPL (2018), 20:1–20:31. https://doi.org/10.1145/3158108

Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*. 65:1–65:10. https://doi.org/10.1145/2603088.2603123

Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*. 187–196. https://doi.org/10.1145/2491956.2491967

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulletin of Symbolic Logic* 5 (1999), 215–244. https://doi.org/10.2307/421090

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI*. 158–174. https://doi.org/10.1145/3453483.3454036

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *PLDI*. https://doi.org/10.1145/3453483.3454031

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. 275–287. https://doi.org/10.1145/2676726.2676985

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL*. 209–220. https://doi.org/10.1145/2676726.2676995

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013). https://doi.org/10.1145/2487241.2487248

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Hongseok Yang. 2007. Relational Separation Logic. *TCS* 375, 1–3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

Jaroslav Ševčík. 2009. *Program transformations in weak memory models.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/3132

Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI*. 306–316. https://doi.org/10.1145/1993498.1993534